LEXICAL ANALYSER AND PASS I OF IITPL COMPILER

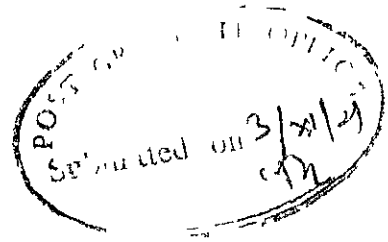
A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

BY

JAGAT KIRAN SINHA

to the

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
November - 1971



CERTIFICATE

Certified that this work on Lexical Analyser and Pass I of IITPL Compiler" has been carried out under my supervision and that this hasnot been submitted elsewhere for a degree.

Dr. H.N. Mahabala
Associate Professor
Department of Electrical Engineering
Indian Institute of Technology
Kanpur

ACKNOWLEDGEMENT

I would like to express my gratitude to Dr. H.N. Mahabala, the supervisor of the project, for his inspiring and valuable guidance.

My special thanks are due to Sri M.K. Sinha, the co-worker of the project who showed his untiring effort and extended his full help for the completion of the project.

But the final achievement still remained a mirage due to the system trouble (specially in DISK) which stood in the way continuously for the last two months.

Finally thanks are also due to Sri K.N. Tewari for his excellent typing.

J.K. Sinha

TABLE OF CONTENTS

SYNOPSIS

This project work has been done for the implementation of IITPL, a language suitable for system implementation, on IBM 7044. This compiler is an out-core type comprising of two passes. Present work elaborates the working of Pass I (of the compiler), which consists of three parts: Lexical Analyser, Syntax Analyser and Pass I Coding of IITPL Source Statements.

Pass I output, either in coding form (i.e. MAP coding); or in string form (for CALL and GOTO etc.) is transferred to PASS II of the compiler for the final coding of the IITPL source statement. Necessary information in the symbol table, ICALL table, GOTO table are also transferred to Pass II to help in the final coding.

The programme is written in FORTRAN IV with Lexical Analyser in MAP, the assembly language of IBM 7044.

CHAPTER I

INTRODUCTION

The purpose of this project "Implementation of IITPL on IBM 7044" is to select and implement a language suitable for system implementation.

This project work has been carried out jointly by the author and Mr. M.K. Sinha. Present thesis deals with the first part of the project. To understand the complete picture, one is suggested to refer the thesis presented by Mr. M.K. Sinha.

For the selection of language for system implementation, the first point of consideration is the level of the language. Here the decision went in favour of high level language, since it has the following advantages:

(1) Machine Independence: The manpower required to implement the system software for a third generation machine is any where from 10 to 50 man years and is a costly affair to start from scratch every time a new machine is designed. Further, users of a particular machine for some time will have developed special software and may not like to reprogramme (or even have the necessary resources) when old machine is being replaced by a new machine. Hence transferring even system software from one machine to other is mandatory from economic considerations. Software written in a high level language

can be machine independent in the sense that it requires small amount of editing and the work on software for the new machine need not start from scratch.

(2) Ease of Updating: Modifications are easy in case of higher level language.

(3) Self-documentating: The programmes written in a higher level language are easy to understand and state the purpose of the programme briefly and lucidly.

(4) Ease of Debugging: Higher level language makes the programme easy for debugging.

(5) Facilitate Thinking: This is due to the fact that higher level language is procedure oriented i.e. one can worry about what to achieve rather than how the machine achieves. When one uses a low level language there is the additional step of synthesizing a logical task in terms of machine operations. Hence one cannot expect to recognise and correct easily logical errors by working with the code in low level language. The programming language used should be goal oriented (in human sense). As to how a machine achieves a logical task should be given minor importance. To get the last bit, literally speaking, by programming in low level language may entail heavy loss of programmer's time and can turn out to be a sure way of inviting hidden disaster.

All the benefits one gets by using a high level language should be of some price - inefficient use of

storage and extended execution time. But still the overall saving is considerable.

Among higher level languages now a days, FORTRAN IV, with SLIP is quite acceptable as an implementation language. But PL/1 (Programming Language/One) is coming up as general purpose higher level language. Besides, almost all the features available in FORTRAN IV, PL/1 has many other features also. Few distinct features are enumerated below:

(1) Storage allocation: Work area for several sub-programmes can be overlapped.

(2) Versatile control commands: Introducing of fairly versatile IF and DO statements increases understandibility and reduces coding.

(3) Character handling: It has ability for string manipulation.

Since a language useful for system implementation, is to be selected, there is no necessity to implement full fledged PL/1. Hence a subset of PL/1 called as IITPL is chosen which has most of the distinct features suitable for system implementation.

The programme has been written in FORTRAN IV with MAP subroutines. The compilation is done in two passes. In this thesis description upto Pass I coding of the compiler is given.

CHAPTER II

IITPL

IITPL is a subset of PL/1 (Programming Language) and consists of many useful facilities as in standard PL/1.

The basic elements necessary to construct the IITPL source statements are constants, Identifiers and Function refcrences, all of which, except labels and keywords, represent numerical quantities.

All these elements mentioned above, except constants, are represented by character strings.

THE CHARACTER SET:

Following characters are valid:

Letters : A to Z

Digits : 0 to 9

Special characters:

Plus	+	Minus	-
Asterisk	*	Left Parenthesis	(
Slash	/	Right Parenthesis	)
Equal Sign	=	Comma	,
Quote	'	Dot	.

Compound Characters:

Semicolon	,.	Exponentiation	**
Colon	..		

Relational Operators:

.EQ. .NE. .GT. .GE. .LT. .LE.

.NG. .NL. .CAT.

Logical Operators:

.AND. .OR. .NOT.

Other characters are not considered to be valid ones by the compiler.

Identifier: An identifier is a single alphabetic character or a string of alphameric characters, not contained in a comment or constant, and preceded and followed by a blank or some other delimeter; the initial character of the string must be alphabetic. The length must not exceed six characters.

Examples:

D56789

A

A12B

All these are valid identifiers.

Keyword: A keyword is an identifier that, when used in proper context, has a specific meaning to the compiler. All the keywords are reserved.

Label: Labels are used to name statements in PL/1. Labels have the general form

name .. Stmt

where name is the label.

Constants: Two types of constants are provided for in IITPL.

(i) Integer Constant: It is a string of one to eleven digits and should be less than or equal to 34359738367.[$2^{35}-1$]

Examples: 20345678001

12

(ii) Binary Constant: A string of "0" and "1", the length of the string should not exceed 36. This type of constant is always represented within quote signs immediately followed by the letter B.

Example: '10101010101111000'B

Array name: An array name represents a sequence of entities. An array element is denoted by the array name followed by a subscript list enclosed in parantheses. Array name itself is any valid identifier. Array name can have multiple subscripts but the number of subscripts should not exceed three.

Subscript: The subscripts of a subscripted (array) name need not be constants. Any expression that yields a valid arithmetic value can be used. If the evaluated value is not an integer, fractional portion is truncated. Truncation is automatically done in IITPL since it purely works in integer mode or binary mode.

Expression: An expression is a representation of a value. A single constant or a variable is an exprcssion.

Combinations of constants and/or variables, along with operators and/or parentheses, are expressions. An expression that contains operators is an operational expression. The constants and variables of an operational expression are called operands.

Examples: A+B

15

B+(C-D)**E

EXPRES

Function Reference Operands: A function reference consists of a name and a parenthesized list of one or more variables, constants, or other expressions. The name is the name of a block of coding written to perform specific computations upon the data represented by the list and to substitute the computed value in place of the function reference.

Statements: An IITPL programme like PL/1 consists of a title (i.e. procedure name), the body of the programme and the END statement.

Continuation card: Only one continuation card is allowed (i.e. maximum of 144 characters is allowed in a source statement).

Comment: It can be any where in the source statement started by "/*" and ended by "*/" in the statement.

Blocks and Scope of the Identifiers

BLOCKS:

A block is delimited sequence of statements that constitute a section of a programme. It localizes names, declared within the block and limits the allocation of variable.

Two kinds of blocks are provided for in IITPL:

(i) PROCEDURE BLOCKS

(ii) BEGIN BLOCKS

The term "Block" refers to either kind of block and the context indicates the kind.

Any block can contain one or more blocks but there can be no overlapping of blocks i.e. a block that contains another block must totally encompass that block.

PROCEDURE BLOCK:

A procedure block, simply called a procedure, is a sequence of statements headed by a PROCEDURE statement and ended by a corresponding END statement.

A procedure block that is contained within another block is called an internal procedure. A procedure block that is not contained within another block is called an external procedure.

Every executable IITPL programme must start with PROCEDURE, which has the MAIN option. Like PL/1 programme

any IITPL programme may consist of one or more external procedures, each of which can contain "local" and "global" identifiers.

General format of declaring Procedure block:

Entry name .. PROCEDURE (parameter ,parameter ...)
data attributes ,.

The first external procedure block must be declared in the following way:

Name.. PROCEDURE OPTIONS(MAIN),.

Procedure block with parameters:

Entry name .. PROCEDURE(parameter [,parameter] ...),.

Functional Procedures:

Entry name .. PROCEDURE(parameter [,parameter ...)])
data attribute,.

Procedure blocks are entered by means of CALL statement or by function references, except the MAIN PROCEDURE from where the normal sequence of execution starts.

The functions of PROCEDURE statement are as follows:

i) It heads a PROCEDURE block.

ii) It defines the primary entry point of the procedure.

iii) It specifies the parameters, if any, for the primary entry point.

iv) It specifies the attribute of the value that is to be returned by the procedure in case of built in functions.

Example:

```
A..PROCEDURE OPTIONS (MAIN),.
   Stmts
B..PROCEDURE,.
   Stmts
   END B,.
C..PROCEDURE (ARG1,ARG2),.
   Stmts
D..PROCEDURE (ARG3,ARG4)FIXED,.
   Stmts
   END A,.
```

In the above example A is an External Procedure as well as Main Procedure block. B, C, D are internal procedures where C is parametric procedure and D is functional procedure.

BEGIN BLOCK:

General format is

[Label..] BEGIN,.

BEGIN blocks are always internal; they must always be contained within another block. The BEGIN statement heads and identifies a BEGIN block. A BEGIN statement is used in conjunction with an END statement to delimit a BEGIN block.

Unlike a procedure block, a label is optional for a BEGIN block. Also unlike procedure block, begin blocks are entered in the normal sequence of execution. Control

can only transfer to the top of begin block.

Example:

```
        ..      ..
        ..      ..
   B1 .. BEGIN,.
         Stmts
         END B1,.
        ..      ..
```

Activation and Termination of BLOCKS

Although both blocks (Procedure and Begin) resemble in many ways but they differ in the way they are activated and executed.

A begin block, like a single statement, is activated and executed in the normal sequence of the programme.

For a procedure, however, normal sequential programme flow passes around the procedure, from the statement before the PROCEDURE statement to the statement after the END statement of that procedure.

Procedure can be activated in the following ways:

i) If it is referred after the keyword CALL in a CALL statement.

ii) As a function reference

Termination of Procedure: A procedure is terminated when one of the following occurs:

i) Control reaches a RETURN statement within the procedure. The execution of a RETURN statement

causes control to be returned to the point of invocation in the invoking block.

ii) Control reaches the END statement of the procedure. Effectively this is equivalent to the execution of a RETURN statement.

iii) The execution of a GOTO statement within the procedure (or any block activated from within that procedure) transfers control to a point not contained within the procedure.

Termination of Begin Block:

A begin block is terminated when any of the following occurs:

i) Control reaches the END statement for the block. When this occurs, next statement after END is executed.

ii) The execution of a GOTO statement within the begin block (or any block activated from within that begin block) transfers control to a point not contained within the block.

iii) Control reaches a return statement that transfers control out of the begin block and out of its containing procedure as well.

Scope of the Identifiers:

Before the scope of the identifiers is to be discussed, let us talk about the Declarative statements provided for in IITPL.

The DECLARE Statement: The DECLARE statement is the principal method for explicitly declaring attributes

of names. No implicit declaration is allowed in IITPL.

General Format:

DECLARE Identifier attribute ...

,Identifier attribute ... ... ,.

Note: The DECLARE statement can only come immediately after a PROCEDURE statement or BEGIN statement or a DECLARE statement. In some cases, several identifiers have the same attributes. In such situations, the attributes can be factored to reduce the amount of writing required. The general form of a DECLARE with factored attribute is

DECLARE (name1, name2, ...) attribute ,.

Attribute specifies the characteristics of the identifier. Following attributes are allowed

FIXED

BIT (n)

where n denotes the number of bits to be used by the corresponding identifier. n should not exceed 36.

FIXED is an attribute which specifies that corresponding identifier is declared as integer variable in the block.

BIT is an attribute which specifies that corresponding Identifier is declared as logical variable.

Examples:

```
DECLARE A FIXED ,.
DECLARE (B(5),C(6,10),D(10,2,3))FIXED ,.
DECLARE AB(10) FIXED ,.
DECLARE BCD BIT(25) ,.
DECLARE (B12(5,6), B35694) BIT(36),.
```

To understand the scope of the identifiers the following example is considered.

```
P1..PROCEDURE OPTIONS (MAIN) ,.
    DECLARE (A,B,C,D,E,F) FIXED ,.
    Stmts
P2..PROCEDURE ,.
    DECLARE (A,B) BIT (20) ,.
    Stmts
P3..PROCEDURE ,.
    DECLARE (A,D) BIT (30) ,.
    DECLARE   E   BIT (25) ,.
    Stmts
B1..BEGIN ,.
    DECLARE (A,F(5,10)) FIXED ,.
    Stmts
P4..PROCEDURE ,.
    DECLARE C(10) FIXED ,.
    Stmts
    END P4 ,.
    Stmts
    END P1 ,.
```

In the previous example identifiers have been declared in different blocks. The external procedure P1 has the following identifiers declared

P1 A,B,C,D,E,F

P2 A,B

In any statement in procedure P2 which refers A or B, A and B will be considered as logical variables (Bit string variables of length 20). If variables among C,D,E and F are referred in P2, these will be treated as global identifiers and will refer to the same variable as in P1, i.e. to say that if any identifier is not declared in a block and is referred in the block, then this will be treated as a global variable and refer to that variable (having the same name) which declared in next higher block. If the referred variable name is not declared in the next higher block, it looks for the next higher block until outermost procedure has been checked. If the referred variable is not found any where, compilation error is given.

In the last example, following is the way in which the compiler will interpret the declaration of variables in different blocks.

P1

name	A	B	C	D	E	F
attribute	FIXED	FIXED	FIXED	FIXED	FIXED	FIXED
name of block if global	-	-	-	-	-	-

P2

Name	A	B	C	D	E	F
attribute	BIT(20)	BIT(20)	FIXED	FIXED	FIXED	FIXED
Name of block if global	-	-	P1	P1	P1	P1

P3

Name	A	D	E	B	C	F
attribute	BIT(30)	BIT(30)	BIT(25)	BIT(20)	FIXED	FIXED
block name	-	-	-	P2	P1	P1

B1

Name	A	F(5,10)	D	E	B	C
attribute	FIXED	FIXED	BIT(30)	BIT(25)	BIT(20)	FIXED
block name	-	-	P3	P3	P2	P1

P4

Name	C(10)	A	F(5,10)	D	E	B
attribute	FIXED	FIXED	FIXED	BIT(30)	BIT(25)	BIT(20)
block name	-	B1	B1	P3	P3	P2

In the above example blocks were nested and each block was having different levels, e.g. P1 with highest level (say 1), P2 with next highest level (say 2) and so on. The level of lowest level block P4 will be 5.

Blocks can have same levels also as shown in the following example:

```
PR1..PROCEDURE OPTIONS (MAIN) ,.
     DECLARE (A,B,C) BIT(36) ,.
     DECLARE D FIXED ,.
     DECLARE (E(5,6,7), F(2,3)) FIXED ,.
     Stmts
PR2..PROCEDURE ,.
     DECLARE (A,C,D) FIXED ,.
     Stmts
     END PR2 ,.
PR3..PROCEDURE ,.
     DECLARE (B,E,F) BIT(27) ,.
     Stmts
     END PR3 ,.
     Stmts
     END PR1 ,.
```

In the above example PR2 and PR3 are of same level (i.e.2). It should be noted that if either of the variables A or C or D is needed in PR3, it will not refer to A,C or D of block PR2 since PR2 and PR3 are of same level. Variable of PR2 cannot be used in PR3 and vice-versa since both are of same level.

Other Statements:

CALL Statement: The CALL statement invokes a procedure and causes control to be transferred to a specified entry point of the procedure.

General Format:

CALL Procedure name [(argument[,argument ...)]] ,.

Syntax Rules:

(i) The procedure name, represents the entry point of the procedure invoked.

(ii) An argument can be a variable, constant or any valid arithmetic expression.

Any procedure, whether external or internal, can always invoke an external procedure (except the containing external procedure) but it cannot always invoke an internal procedure that is contained in some other procedure. Those internal procedures that are at the first level of nesting relative to a containing procedure can always be invoked by that containing procedure, or by each other.

Example:

```
P1..PROCEDURE ,.
    Stmts
P2..PROCEDURE ,.
    Stmt P2-1 ,.
    Stmt P2-2 ,.
P3..PROCEDURE ,.
    Stmt P3-1 ,.
    Stmt P3 2 ,.
    END  P2 ,.
    Stmts
P4..PROCEDURE ,.
    Stmt P4-1 ,.
    END P4 ,.
    END P1 ,.
```

In this example P1 can invoke procedures P2 and P4 but not P3. It can be easily seen that P2 and P4 are those internal procedure which are at the first level of nesting relative to the containing procedure P1.

Also P2 and P4 can invoke each other. P2 can invoke P3 also. However, P4 cannot invoke P3 and vice-versa. Here it differs from PL/1. In PL/1 P3 can invoke P4 while in IITPL it is not valid.

DO Statement:

Function: The DO statement heads a DO-group and can also be used to specify repetitive execution of the statements within the group.

General Format:

DO c.v. = Specification,specification ... ,.

where c.v. denotes the control variable and specification has the form

$$\text{Expression 1}\left[\begin{matrix}\text{TO expr.2 BY expr.3}\\ \text{BY expr.3 TO expr.2}\end{matrix}\right]\left[\text{WHILE(expr.4)}\right]$$

In simple way it can be said that the specifications may take one of the following forms or combination of one or more of the following forms:

(a) expr.1, expr.2,..., expr.n

(b) expr.1 BY expr.3 TO expr.2

(c) expr.1 TO expr.2 BY expr.3

(d) WHILE (condition)

Any DO group is terminated by an END statement. Actually the DO statement, the END statement and any statement in between comprise a DO group.

DO group have the following characteristics:

(i) The expressions in the specifications are evaluated only once, at the time the DO statement is first encountered. Thus the expression values cannot be effectively modified by statements within the group. However the control variable can be modified. If such modifications give it a value outside the specified range, the loop will not be repeated after the current pass is completed.

(ii) A GOTO statement within the scope of a DO loop may transfer control to a point outside the loop, thus terminating the loop prematurely.

(iii) A GOTO statement outside a DO group must not transfer to a point within the loop. (A GOTO statement may, however, transfer to a point within a DO group).

(iv) The value of the control variable, after control has left a DO loop, depends upon the conditions under which the loop was left, i.e. to say

If the loop is terminated normally, the control variable has the first value which fails to meet the conditions for execution of the loop.

If the loop is terminated prematurely, e.g. by a GOTO statement which transfers out of the loop, the value of control variable is the value it had when the GOTO statement was executed.

Example:

```
PRTAIN..PROCEDURE OPTIONS (MAIN) ,.
        DECLARE (A(10), I) FIXED ,.
        GET EDIT (A(10) DO I = 1 TO 10)(10 F6) ,.
  LAB1..DO I = 1 TO 10 BY 1 ,.
        IF (A(I) = 5) GOTO LAB2 ,.
        END LAB1 ,.
  LAB2..PUT EDIT (I) (F6) ,.
        END PRTAIN ,.
```

If DO group ends normally, I will have a value 11 when it comes out of DO group. Otherwise I will have that value corresponding to which A(I) = 5. If there are many elements in A which have the value 5, first one will be taken which transfers control to LAB2.

END Statement:

Function: The END statement terminates blocks and groups.

General format: END Label ,.

General rules:

i) If a label follows END, the statement terminates the unterminated group or block headed by the nearest preceding DO, BEGIN or PROCEDURE statement having that label. It also terminates any unterminated groups or blocks physically within that group or block.

ii) If a label does not follow END, the statement terminates that group of block headed by the nearest preceding DO, BEGIN or PROCEDURE statement for which there is no corresponding END statement.

iii) If control reaches an END statement for a procedure, it is treated as a RETURN statement.

Example:

```
P1..PROCEDURE ,.
    Stmts
B1..BEGIN ,.
    Stmts
D1..DO I = 1 TO 8 ,.
    Stmts
    END ,.
    END P1 ,.
```

In the above example first END statement ends the DO group since DO group is the nearest one. While second END statement terminates P1 as well as B1 since B1 is contained in P1.

IF and ELSE Statement:

The IF statement tests the value of a specified expression and transfers control according to the result of that test.

General format: IF element expression
THEN unit-1
ELSE unit-2.

Syntax Rules:

i) Each unit is a single statement (except DO, END, PROCEDURE, BEGIN, DECLARE, FORMAT).

ii) The IF statement itself is not terminated by a semicolon, however each unit specified must be terminated by a semicolon.

iii) Unit should not be a labelled statement.

General Rules:

1) The element expression is evaluated and converted to a bit string. If ELSE cause is specified then following action is taken:

If any bit in string is 1 it executes unit-1 and then transfers control to the statement following the IF statement otherwise unit-2 is executed and unit-1 is skipped and control passes to the next statement.

When ELSE cause is not specified, following actions are taken:

If any bit in the converted bit string is 1, unit-1 is executed and control passes then to the next statement. If all bits in the string are zero, unit-1 is skipped and control passes to the next executable statement.

ii) IF statement may be nested i.e. either "unit" (unit-1 or unit-2) or both, may itself be an IF statement.

Examples:

(1) IF condition 1 THEN Statement-1 ,.
ELSE IF condition 2 THEN statement-2 ,.
... ...
ELSE IF condition n THEN statement-n ,.
ELSE statement-m ,.
next statement ,.

To analyse the statements mentioned in previous page, let us examine it from bottom to top, step by step:

i) Statement-m will be executed if condition n is false. If it is true statement-n will be executed.

ii) Condition n will be evaluated if condition 2 is false. If condition 2 is true, statement-2 will be executed, i.e. to say IF - ELSE construction always presents mutually exclusive alternatives.

iii) Similarly condition 2 will be evaluated if condition 1 is false.

Thus it can be said that

ELSE, if used, specifies an alternative path for the last preceding IF for which an ELSE path is not already specified.

Example 2: Another type of nesting of IF statement is as follows:

```
IF condition-1 THEN
   if condition-2 THEN statement-2
   ELSE statement 2a ,.
   ELSE statement-1a ,.
   next statement,.
```

Example 3:

```
IF condition-1 THEN statement-1 ,.
   ...    ...    ...
   ELSE IF condition-n THEN statement-n ,.
   next statement ,.
```

In example 3 there is no alternative path (ELSE statement) for the last IF. However, if condition-n is true statement-n will be executed and control will then pass to the next statement. If condition-n is false it will skip the statement-n and transfer control to next statement.

GOTO Statement:

Function: The GOTO statement causes control to be transferred to the statement identified by the specified label.

General format :

$$\left\{\begin{matrix}\text{GO TO}\\ \text{GOTO}\end{matrix}\right\} \quad \left\{\begin{matrix}\text{Label constant ,.}\\ \text{Element label variable ,.}\end{matrix}\right\}$$

General rules:

i) If an "element label variable" is specified, transfer of control depends upon the value of "element label variable" at the time of execution of GO TO statement. Since value may change, hence it is not essential that it will always transfer to the same label.

ii) A GOTO statement cannot pass control to an inactive block.

iii) A GOTO statement cannot transfer to a point within a DO group from outside the group.

iv) If a GOTO statement transfers control from within a block to a point not contained within that block the block is terminated. Also, if the transfer point is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are also terminated.

v) When a GOTO statement transfers control out of a procedure that has been invoked as a function, the evaluation of the expression that contained the corresponding function reference is discontinued.

Example:

```
P1..PROCEDURE OPTIONS (MAIN) ,.
L1..Stmt ,.
...   ...
L2..Stmt ,.
...   ...
    GOTO L2 ,.                          (6)
    CALL P3 ,.
    ...   ...
    CALL P2 ,.
P2..PROCEDURE ,.
...   ...
    CALL P3 ,.
L1..Stmt ,.
B1..BEGIN ,.
    Stmts
B2..BEGIN ,.
L1..Stmt ,.
    Stmts
B3..BEGIN ,.
...   ...
    GOTO L1 ,.                          (1)
    ...   ...
    GOTO L2 ,.                          (2)
    ...   ...
    GOTO L3 ,.                          (3)
```

```
L3..END B3 ,.
    ...   ...
    END B2 ,.
L2..Stmt ,.
    ...   ...
    END B1 ,.
    END P2 ,.
P3..PROCEDURE ,.
    ...   ...
    GOTO L1 ,.                    (4)
    ...   ...
    GOTO L2 ,.                    (5)
    ...   ...
    END P1 ,.
```

In the above example statement GOTO L1, numbered (1) transfers the control to the label L1 in block B2 and block B3 is terminated, since there is no label L1 in block B3; B2 is the immediate outer block.

Statement GOTO L2, numbered (2) transfers the control to label L2 in begin block B1 and blocks B3 and B2 are terminated since L2 label is present neither in B2 nor in B3. But statement GOTO L3, numbered (3) and GOTO L2 numbered (6) do not terminate any block since labels are in the very block and control transfers to the corresponding label.

Now statements numbered (4) and (5) are to be given special attention, since their meaning varies according to the activation of the block.

Case I: If P3 has been called from procedure P1 (i.e. P2 is inactive) in that case - statement, GOTO L1, numbered (4), will transfer control to label L1 in procedure block P1 and P3 will be terminated. Similarly statement, GOTO L2, numbered (5) will transfer control to label L2 in block P1; and P3 will be terminated.

Case II: If P3 has been called from procedure P2 (i.e. all procedures P1, P2 and P3 are active) statement, GOTO L1, numbered (4), will transfer control to label L1, in procedure P2 and P3 will be terminated since P2 has called P3 and label L1 is there in P2. Whereas statement, GOTO L2, numbered (5), will transfer control to label L2 in procedure P1 and both P3 and P2 will be terminated since L2 is neither in P3 nor in P2.

RETURN Statement

Function: The RETURN statement terminates execution of the procedure that contains the RETURN statement. It may also return a value if it is written within a functional procedure.

General format:

Option 1 RETURN ,.

Option 2 RETURN (expression) ,.

General Rules:

i) Only, the RETURN statement in option 1, can be used to terminate procedures, not invoked as function procedures. Control is returned to the point logically following the invocation.

ii) The RETURN statement in option 2 is used only to terminate a procedure invoked as a function procedure. Control is returned to the point of invocation, and the value returned to the function reference is the value of the expression specified.

iii) If control reaches an END statement corresponding to the end of a procedure, this END statement is treated as a RETURN statement (of the option 1 form) for the procedure. There may be any number of RETURN statement in a procedure.

Example:

```
A.. PROCEDURE ,.
    DECLARE (C,D) FIXED ,.
   CALL B (C,D) ,.
    C = C + D + FUN (C) ,.
    END A ,.
B.. PROCEDURE (X,Y) ,.
    DECLARE (X,Y) FIXED ,.
    IF X = 0 THEN GOTO LAB1 ,.
    X = X + 2 ,.
    Y = X**2 ,.
    RETURN ,.
```

```
LAB1..X = 5 ,.
      Y = 4 ,.
      END ,.
FUN..PROCEDURE (X) FIXED ,.
      DECLARE X FIXED ,.
      Strts
      RETURN (X **3 + X/2 + 3) ,.
      END   ,.
```

I/O Statement: Only EDIT directed I/O statements are allowed in IITPL.

GET/PUT Statement:

Function: The GET statement is a STREAM transmission statement that is used for assignment of data from an external source to one or more internal receiving fields (i.e. one or more variables).

The PUT statement is a STREAM transmission statement that is used for assignment of data from one or more internal fields (i.e. one or more variables to an external source).

General Format:

GET/PUT [option list] EDIT (data list) (format list)
[(data list) (format list)] ,.

Following is the format of

i) "Option list": FILE (file name)

File names are TAPE0, TAPE1, TAPE2, TAPE3 and TAPE4. If no option list is there, card reader is taken as the external source in case of GET statement; and printer in case of PUT statement.

ii) "Data list": element ,element ...

(a) A datalist must always be enclosed in parentheses.

(b) The element can be one of the following:

An element, array, or a repetitive specification (similar to a repetitive specification of a DO group).

Example: (A,B(4,5),(C(I,4)DO I=2 TO J BY K))

Unlike PL/1, only constant and unsubscripted variables are allowed for DO specifications for I/O statements.

iii) "Format list":

$$\left\{\begin{array}{l}\text{item}\\ \text{n item}\\ \text{n(format list)}\end{array}\right\}\left[\begin{array}{l},\text{item}\\ ,\text{n item}\\ ,\text{n(format list)}\end{array}\right]$$

(a) Each item represents a format item as described below.

(b) The letter n represents an iteration factor which is unsigned decimal integer constant. A blank or left parenthesis must separate the iteration factor from the format item.

(c) There are three types of format items;

(I) Data format item

(II) Control format item

(III)Remote format item.

Data format Items:

Fixed point (F), character string (A)

F[(W)], A[(W)]

The letter W, an unsigned decimal integer constant represents the number of characters in the field.

Control format items:

The control format items are the spacing format item (X) and SKIP format items

X(W)

SKIP (W)

Here W has the same meaning as above.

Remote format item: R(1)

The letter l specifies the label of a format statement, located elsewhere. Unlike PL/1, l cannot be a label variable.

FORMAT Statement: Purpose:

The FORMAT statement specified a format list that can be used by edit directed transmission statements to control the format of the data being transmitted.

General format: Label: FORMAT(format list)

Syntax rules:

i) Unlike PL/1, multiple labels are not allowed, but "label" must be specified for a FORMAT statement.

ii) The format list here is same as discussed in I/O statement but remote format item must not be in the format list of a format statement.

OPEN Statement:

Function: The OPEN statement opens the file of given file name. I/O function with the file can only be done if the file is open already. If file is already open, it sets the pointer again at the beginning point.

General format: OPEN FILE (file name) [,FILE (file name)]...,.

where "file name" is one of the following:

i) TAPE0 ii) TAPE1 iii) TAPE2 iv) TAPE3 v)TAPE4

"file name" must be enclosed by the parentheses.

CLOSE Statement:

Function: The CLOSE statement closes the file of given file name. I/O functions cannot be done with a closed file.

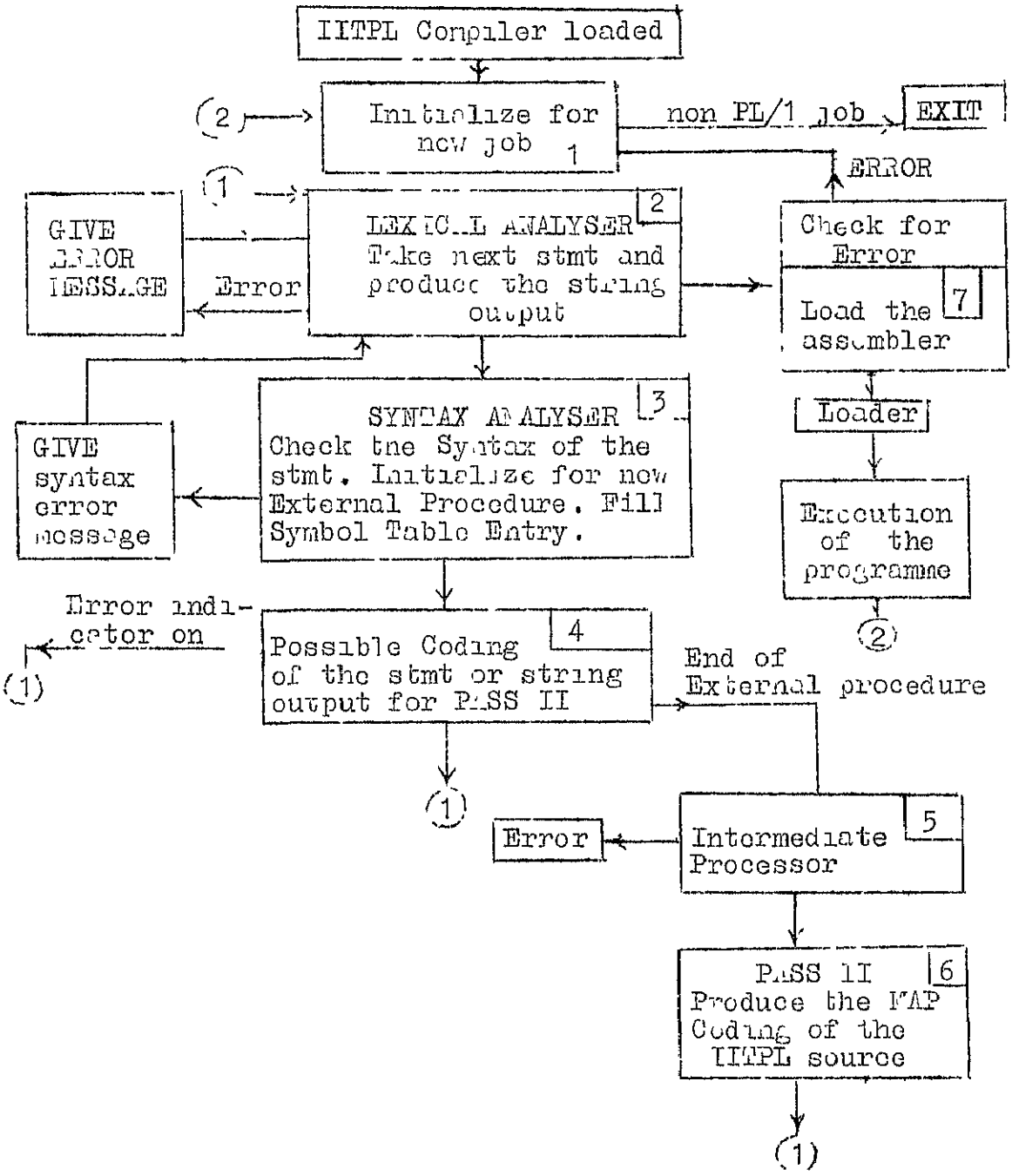
General format: CLOSE FILE(file name) [,FILE(file name)] ...,.

where "file name" has the same meaning as discussed in OPEN statement.

Unlike PL/1, no option is allowed with OPEN/CLOSE statements.

FLOW OF CONTROL IN THE IITPL COMPILER

(Basic structure of the compiler)



CHAPTER III

BASIC STRUCTURE OF THE COMPILER AND SYMBOL TABLE:

Present chapter gives a brief description of the different blocks of the IITPL compiler implemented at IIT on IBM7044. A detailed description of the symbol table entries for various types of name is also presented here.

Basic structure of the Compiler: (Lexical, Syntax and Coding): Adjoining figure shows the flow of control in the IITPL compiler implemented at IIT on IBM7044. Each block has been given a number put in the corner of the rectangle.

(1) First block sets the initial values of certain variables (indicators) whenever new job starts.

(2) Second block named as Lexical analyser takes one IITPL statement at a time, checks for any invalid character and other possible errors. If no error is found, it produces the string output to be used by the Syntax Analyser. It also squeezes out the redundent blanks and comment portion of the source statement. A detailed description is given in the next chapter.

(3) Syntax Analyser: As the name suggests, it checks the syntax of the source statement (now in string form produced by the Lexical Analyser). It first finds out the

type of the statement and transfers control to the corresponding section of the programme which checks for the validity of the statement. Initializations are done for each programme segment (i.e. for each External Procedure Block). Besides this some initializations for each subprogramme or block (Procedure or BEGIN Block) are also done. Symbol table entries are filled for various types of name e.g. procedure name, variable name and label name. After the statement being found syntactically correct, it transfers control to the block number 4 (in the diagram) for possible coding or string output. Error messages are given to syntactically incorrect statement and error indicator is set on.

(4) Possible coding of the statement

or

String output for Pass 2

In this section of the programme, it is checked if error indicator is on. If yes, it exists and control goes back for taking next statement. Otherwise possible coding is done for the new statement. By possible coding, we mean, that only those source statements, which do not require any further information for coding, are coded in Pass 1. Arithmetic assignments, IF, DO etc. are examples for this. Few statements viz. PROCEDURE and CALL cannot be coded in Pass 1. GOTO statement also cannot always be coded in Pass 1. In such cases this section of the programme produces a syntax string output which is used by Pass 2.

We shall go into the details of these things in Chapters V and VI.

(5) Intermediate Processor: It checks for the validity of subprogramme linkage (internal procedure block linkage). It also does modifications in GOTO table (prepared for, checking any invalid transfer and for keeping the information concerning valid transfer). This modification is done with the help of ICALL table which stores all the cross references between blocks (refer to the thesis presented by Mr. M.K. Sinha).

(6) Pass 2: This section scans from the beginning of the programme segment and puts the coding done by Pass I in a card reflection. If string output produced by Pass I is found, it does the MAP coding of this string in a card reflection. It also checks for any illegal transfer in the programme segment (External Procedure). At the end of Pass 2, Symbol table is erased.

(7) Loading of the Assembler: This section checks for any error. If error is found, it terminates the job and goes back to the initial point for next job. Otherwise Assembler is loaded with produced MAP coding of IIIPL source as its input. Finally the output of the assembler with runtime routines are made input to the Loader. $ENTRY is also generated in this section and *DATA is replaced by this $ENTRY.

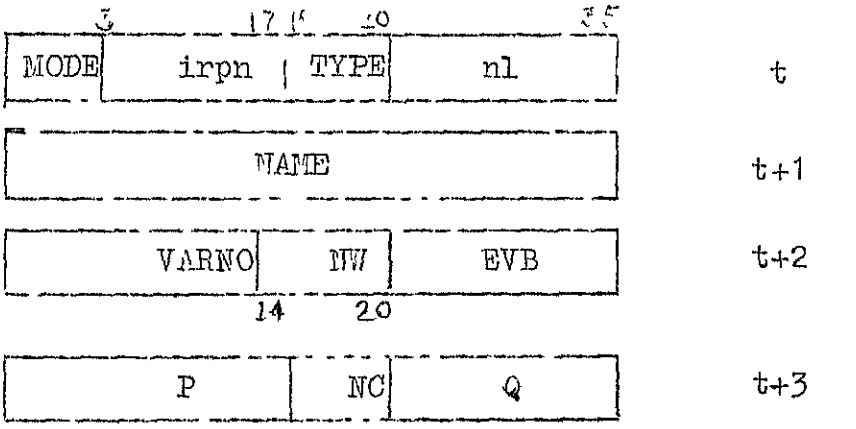
THE SYMBOL TABLE

During the Pass 1 of the compilation of a source programme, Symbol table entries are filled. A Symbol table entry consists of four consecutive words packed with information concerning one of the following entities which occur in the programme segment (External Procedure) being compiled:

(1) Variable name
  (i) Non subscripted
  (ii) Single subscripted
  (iii) Double subscripted
  (iv) Tripple subscripted
(2) Procedure name
(3) Label name

Symbol Table Entry for

(a) Variable name:



Location

t

t+1

t+2

t+3

nl = pointer to next item with the same hash address if there is any. Else nl = 0.

TYPE = 0 for non subscripted variables.

= 1 for single subscripted variables.

= 2 for double subscripted variables.

= 3 for triple subscripted variables.

= 4 for Procedure name.

= 5 for label name.

EVB: This is the Symbol table address of the block under which this name (variable, procedure or label) has come. EVB=0 for the External Procedure name.

MODE = 1 for attribute FIXED.

= 2 for attribute BIT.

= 3 for attribute CHARACTER.

irpn = Internal relative position number of the variable in the block under which this variable name has been declared.

VARNO= Internal sequence number of the subscripted variable.

= 0 for non subscripted variable.

NW = Number of words needed by the variable.

P = Upper limit of the first subscript.

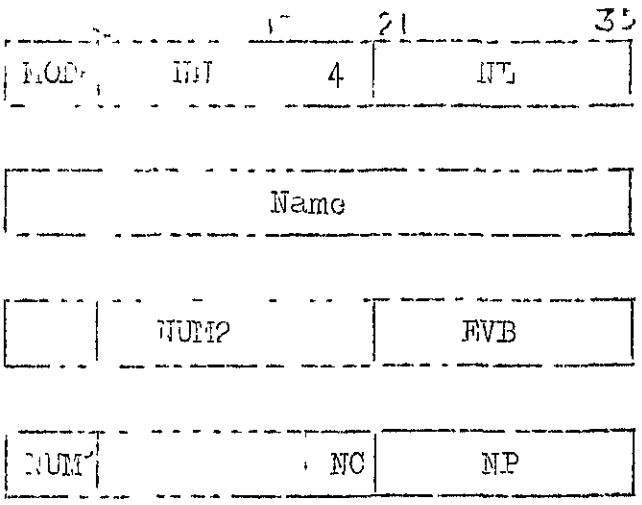
= 0 for non subscripted variable.

Q = Upper limit of the second subscript.

= 0 for single and non subscripted variable.

NC = Number of characters in the variable name itself, e.g. if the variable is 'ABD' then NC=3.

(b) Procedure name:



MODE = 1 for simple procedure without any formal parameter.

= 2 for parameter procedure.

= 3 for functional procedure with attribute FIXED.

= 4 for functional procedure with attribute BIT.

= 5 for functional procedure with attribute CHARACTER.

NN = Number of bits needed for a BIT or CHARACTOR attribute of Built in function or Home made function.

= 0 otherwise.

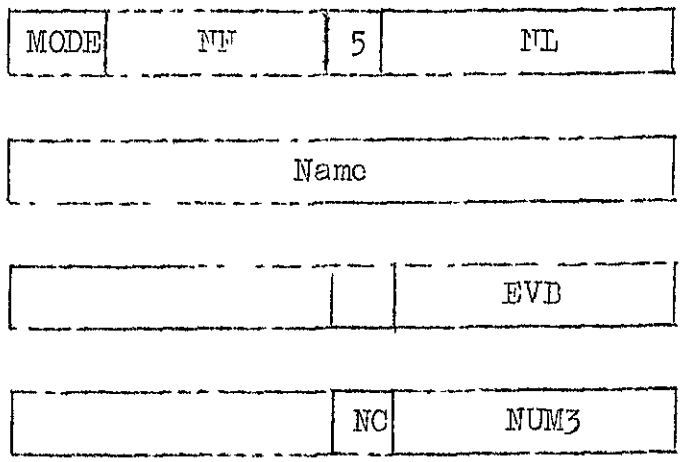
NUM2 = Level of the block.

NP = No. of formal parameters in the procedure if any.

= 0 otherwise.

(c) Label name:

Label name includes DO name and BEGIN name also.



MODE = 0 if this label is under Procedure block.

= 1 if it is under Begin block.

= 2 if it is under DO block.

NN = Level sequence number.

NUM3 = Internal DO number if this label is under DO block.

CHAPTER IV

LEXICAL ANALYSER

The lexical analyser, the first step of compilation, takes the following actions for each IITPL statement taken from input source.

1) It removes the comment portion from the statment, if any.
2) It squeezes out the blanks.
3) It gives information on encountering control cards (i.e. *PL/1, *DATA).
4) It gives error if there is any invalid string or symbol or illegal character (Details of errors are given in Appendix I).
5) It produces a string output for each IITPL statement. Along with the string, one statement characteristic-word is also provided.

Only one continuation card is allowed.

The content of source card of each IITPL statement is parsed character by character and the string is produced, any element of which will be one of the following entities:

i) Symbol with its characteristics.
ii) Constant (Integer, bit or character) with their characteristics.
iii) Operators or delimiters.

i) SYMBOL and its characteristic:

In lexical string each symbol is represented by a pair of words. First word is the characteristic-word which stores the number of characters of the symbol and the second one is the symbol-word which stores the symbol itself.

A symbol is a string of one to six alphanumeric characters with first character as alphabetic. No attention is given to differentiate Keywords, labels and identifiers. All are treated equally, but some keywords are of more than six characters. So, any symbol, whose number of characters exceeds six, is checked for a possible keyword. If yes, an internal name is given to it with first character as special character (↑) to differentiate with other symbols, otherwise error indicator is put on.

Example:

	SYMBOL		CHARACTERISTIC WORD	SYMBOL WORD
1)	AB	-	0 0 0 0 0 2	A B ∅ ∅ ∅ ∅
2)	C4D8	-	0 0 0 0 0 4	C 4 D 8 ∅ ∅
	KEYWORD (Character 6)			
1)	PROCEDURE	-	0 0 0 0 0 2	↑ 2 ∅ ∅ ∅ ∅
2)	DECLARE	-	0 0 0 0 0 2	↑ 3 ∅ ∅ ∅ ∅

ii) CONSTANT and its characteristic word:

Similar to SYMBOL, each constant (integer, bit or character) has a characteristic word consisting of the information given below:

CHARACTERISTIC WORD - | MODE | LENGTH | TYPE | NW |

Length - Number of characters in the constant

NW - Number of words following the characteristic word in which the constant is kept.

MODE = 0 For symbols.(as we have already seen).

= 1 For Integer,2 for Bit,3 for Chtr.constant.

= 4 For operators and delimiters (discussed later).

TYPE = 0 For FIXED DECIMAL INTEGER CONSTANT

= 1 For CHARACTER STRING CONSTANT

= 2 For BIT STRING CONSTANT

This characteristic word is followed by the constant word.

CONSTANT -

a) INTEGER - A string of from one to eleven numeric characters is an integer. Any blank or any operator delimits the string. The integer is kept in octal form in the word following characteristic word.

Example:

	CONSTANT	CHARACTERISTIC WORD						CONSTANT					
1)	25	8	0	2	0	0	1	0	0	0	0	3	1
2)	32768	8	0	5	0	0	1	0	0	0	8	0	0

b) BIT - String of binary digits (0 or 1) under quote signs(') followed by alphabet B with maximum number of 36 binary digits is taken as bit string and is kept in second word bitwise and left justified.

Example: 1) '0101C'B

+	0	5	+	0	1	D	0	0	0	0	0

c) CHARACTER String of characters (60 character set) under quote signs followed by blank or any delimiter (except ') with maximum of 36 characters is taken as character string. Two continuous quote signs mean one quote sign as an element of the string. If number of characters is more than 6, it takes more than one word.

Examples:

CONSTANT CHARACTERS IN ... CONSTANT STRING

1) 'ABC' -

H	0	3	b	0	1	A	B	C	∅	∅	∅

2) 'AD"$CF'

H	0	6	b	0	1	A	D	'	$	C	F

3) 'AMN*@KL('

H	0	8	b	0	2	A	M	N	*	@	K	L	(	∅	∅	∅	∅

iii) Operators and delimiters - All arithmetic operators (+,-,* etc.), compound arithmetic operator (**), logical operators (.AND., .OR., etc.) and delimiters (Colon "..", Semicolon ",.", Comma ",") are given internal code. There is no characteristic word for these, since the internal code itself keeps all the informations. C(operator)

keeps the mode (=4) to differentiate it with other types of string element.

Example: 1) .AND. -

-	0	=	0	0	0

2) .. -

-	∅	∅	∅	.	.

In the statement-characteristic-word, following information are kept:

i) Statement has a label or not.

ii) Statement is non-executable or not.

iii) Length of the string output.

S 1 2 STATEMENT-CHARACTERISTIC-WORD

		ICNTRL	LENGTH

LENGTH - length of the string

S is on - statement has a label

1 is on - Declare keyword is present

2 is on - Procedure keyword is present

ICNTRL = 1 - 'PL/1 encountered.

= 2 - 'DATA encountered.

A lexical output of IITPL statement is shown below:

IITPL Statement Lexical string

/'ABC*/A..∅∅∅C=12,.

-	0	0	0	0	9
-	∅	∅	∅	.	.
-	0	2	0	0	0

0	0	0	0	0	1
0	0	0	0	0	1
8	0	2	0	0	1

A	∅	∅	∅	∅	∅
C	∅	∅	∅	∅	∅
0	0	0	0	0	1

CHAPTER V

SYNTAX ANALYSER AND BASIC ROUTINES

Syntax Analyser checks the syntax of the source statement (now in the string form which has been procuced by the lexical analyser) according to the general format of the statement given in Chapter II. It first finds out the type of the statement and transfers control to the corresponding section of the programme which checks for the validity of the statement. PROCEDURE and DECLARE are the keywords of more than six characters and, hence, in lexical analyser itself, PROCEDURE and DECLARE statements are given special characteristics which make syntax checking easy for the Syntax Analyser. If any labelled statement is found, the label is searched in the GOTO table. If this label is found in GOTO table as a proper label for transfer, it is erased from this table.

Initializations are done for each external procedure. Besides this, some initializations are done for the internal blocks. Symbol table entries are filled and used during syntax checking.

How the flow of control transfers to the corresponding routine, is shown in the flow chart named as "MAIN".

Before a brief description of the basic routines is given, let us talk about the some useful variable names:

COLECT: This is a single subscripted integer variable with its subscript having the upper bound as 144. Actually this is the area where the lexical string output for a new source statement is put.

ICHAR: It is the characteristic word for each source statement put by the lexical analyser.

N: It is a pointer to the COLECT area which is used to refer any word in the COLECT area.

The descriptions of the basic routines used by the compiler for the syntax checking are given below:

All these routines are written in FORTRAN IV.

IXTVAR: This is a table for external variable (global identifiers) which is filled by the symbol table routine.

SYMTAB: This is the symbol table routine, which calculates the hash address of the key (passed into it through the variable name KEY by COMMON statement), finds its linkage, searches the key and calls the routine PUTINF if the key is to be filled. If the key is sent only for searching, it searches and gives message accordingly.

Here uses of some indicators and how SYMTAB interprets it, are given:

1) LINK=0, PPRNAM=0

It means that the key is a variable name, and has been declared with its attribute and to be entered in SYMTAB.

ii) LINK=1, PPRNAM=0:

Search this name in symbol table.

iii) LINK=0, PPRNAM=1:

It means that the symbol table has been called by LOADR to fill the arguments in SYMTAB.

iv) LINK=1, PPRNAM=1:

It is a formal parameter (which has already been called) of a procedure and only other information is to be stored therein.

After returning from SYMTAB routine, if

i) SYMERR≠0 means there is an error in the source statement.

ii) FOUND=.TRUE. means that the key is found in the current block.

iii) FOUND=.FALSE.

(a) VALUE=0 means the key is not found.

(b) VALUE≠0 means the key is found in the higher level active block. In case of variable name VALUE points the entry in EXTVAR while in other case it points the entry in symbol table.

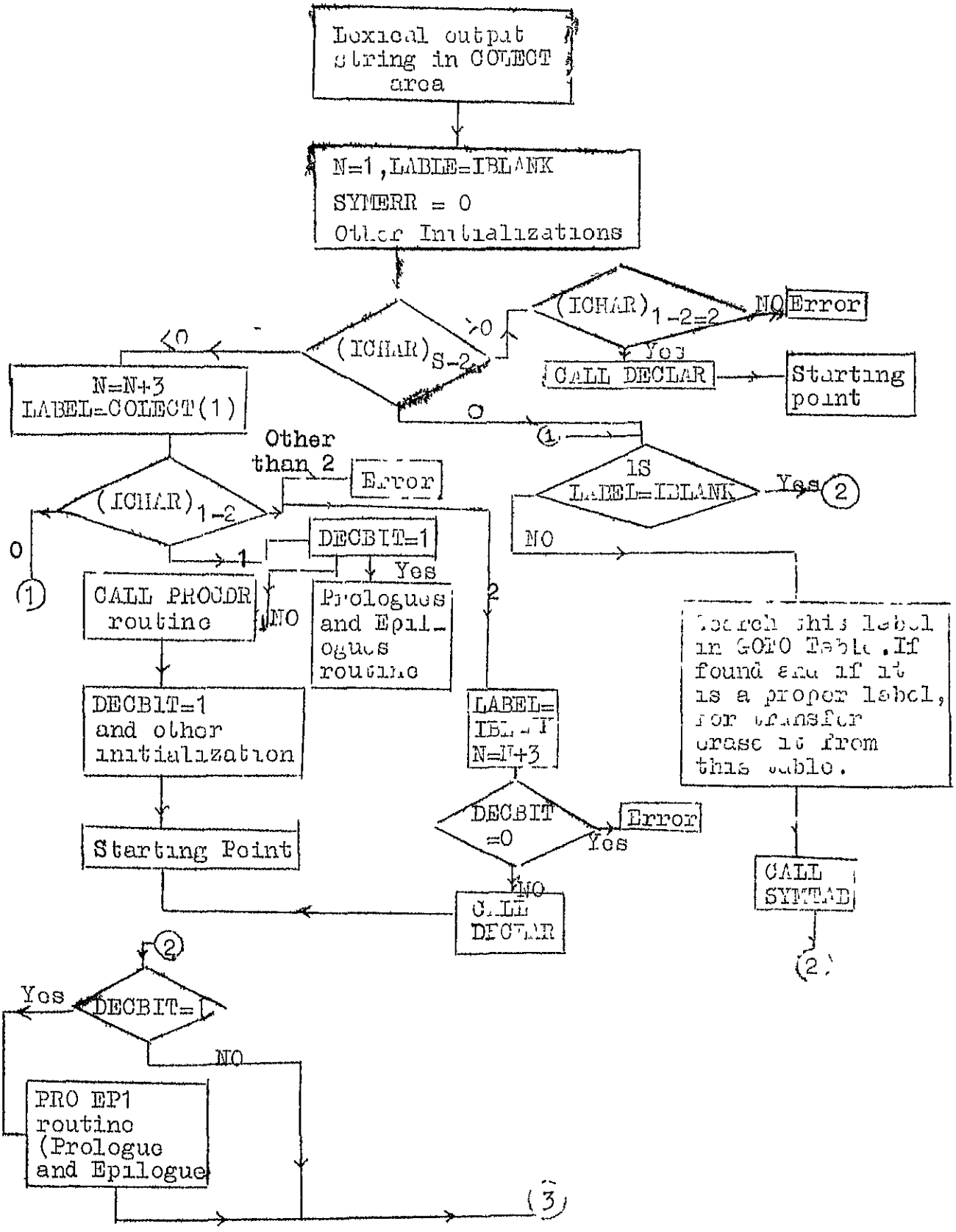
PUTINF: This routine when called from SYMTAB, puts the the information in FSL (free storage location of the symbol table). It is called only from SYMTAB in case of

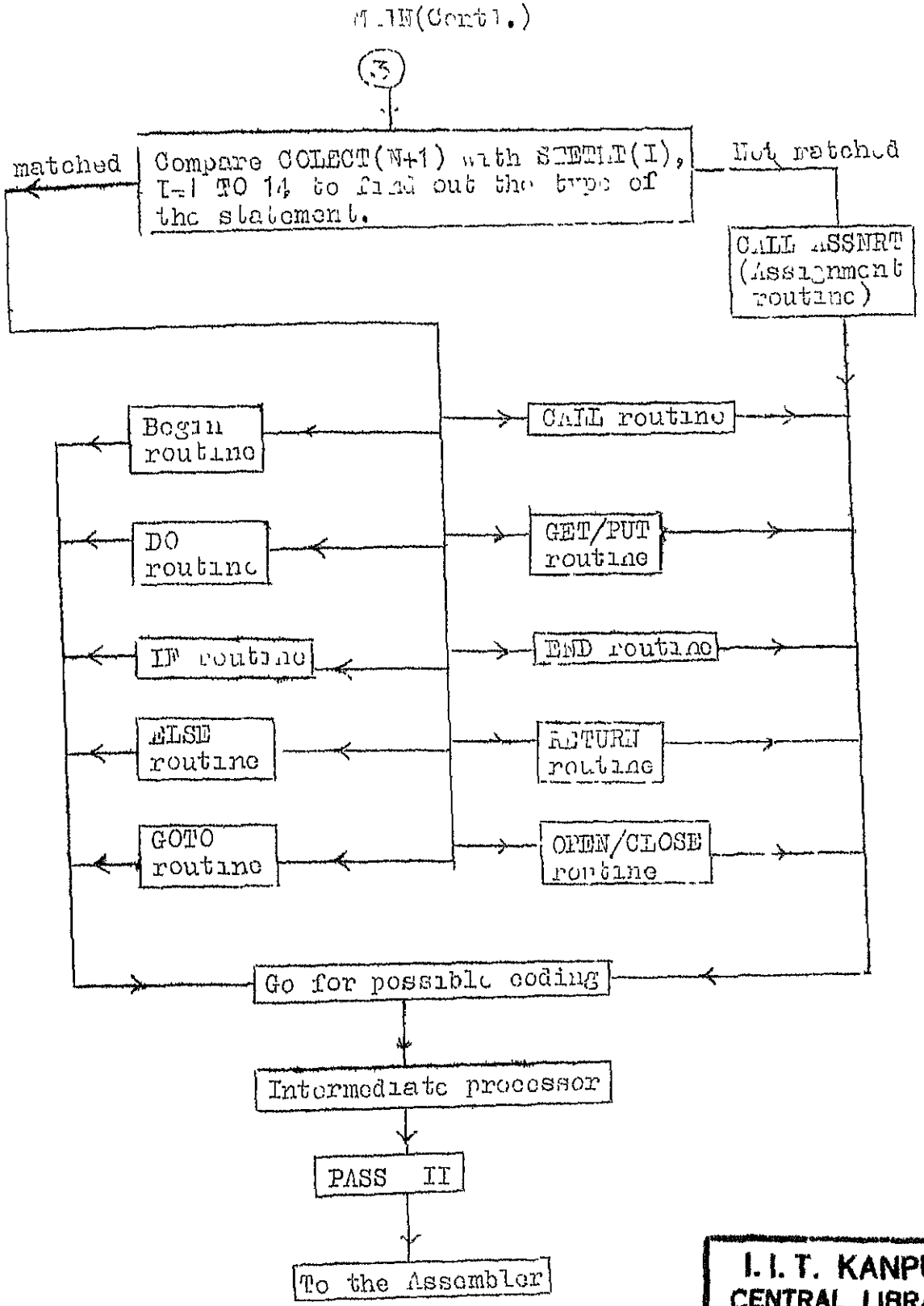
i) LINK=0

ii) LINK=1 and PPRNAM=1.

In the first case, it puts the information in the next free location of FSL, while in the second case, it

MAIN


Lexical output string in COLECT area
N=1, LABLE=IBLANK
SYMERR = 0
Other Initializations
(ICHAR)S-2
<0
>0
0
(ICHAR)1-2=2
NO
Error
Yes
CALL DECLAR
Starting point
N=N+3
LABEL=COLECT(1)
(ICHAR)1-2
Other than 2
Error
0
1
2
DECBIT=1
Yes
NO
CALL PROCDR routine
Prologues and Epilogues routine
DECBIT=1 and other initialization
Starting Point
IS LABEL=IBLANK
Yes
NO
LABEL=IBL-1
N=N+3
DECBIT =0
Error
Yes
NO
CALL DECLAR
Search this label in GOTO Table. If found and if it is a proper label, for transfer erase it from this table.
CALL SYMTAB
(2)
(3)
DECBIT=1
Yes
NO
PRO EP1 routine (Prologue and Epilogue



puts the information in the area of FSL, which is pointed by SYMTAB routine.

DECLAR: This routine is called whenever a DECLARE statement is encountered. This routine checks the syntax of DECLARE statement and calls the routine SYMTAB to fill the entries in the symbol table for declared variables. It also does the coding for the array name. The basis of calling SYMTAB routine is the same as mentioned earlier in the description of SYMTAB routine.

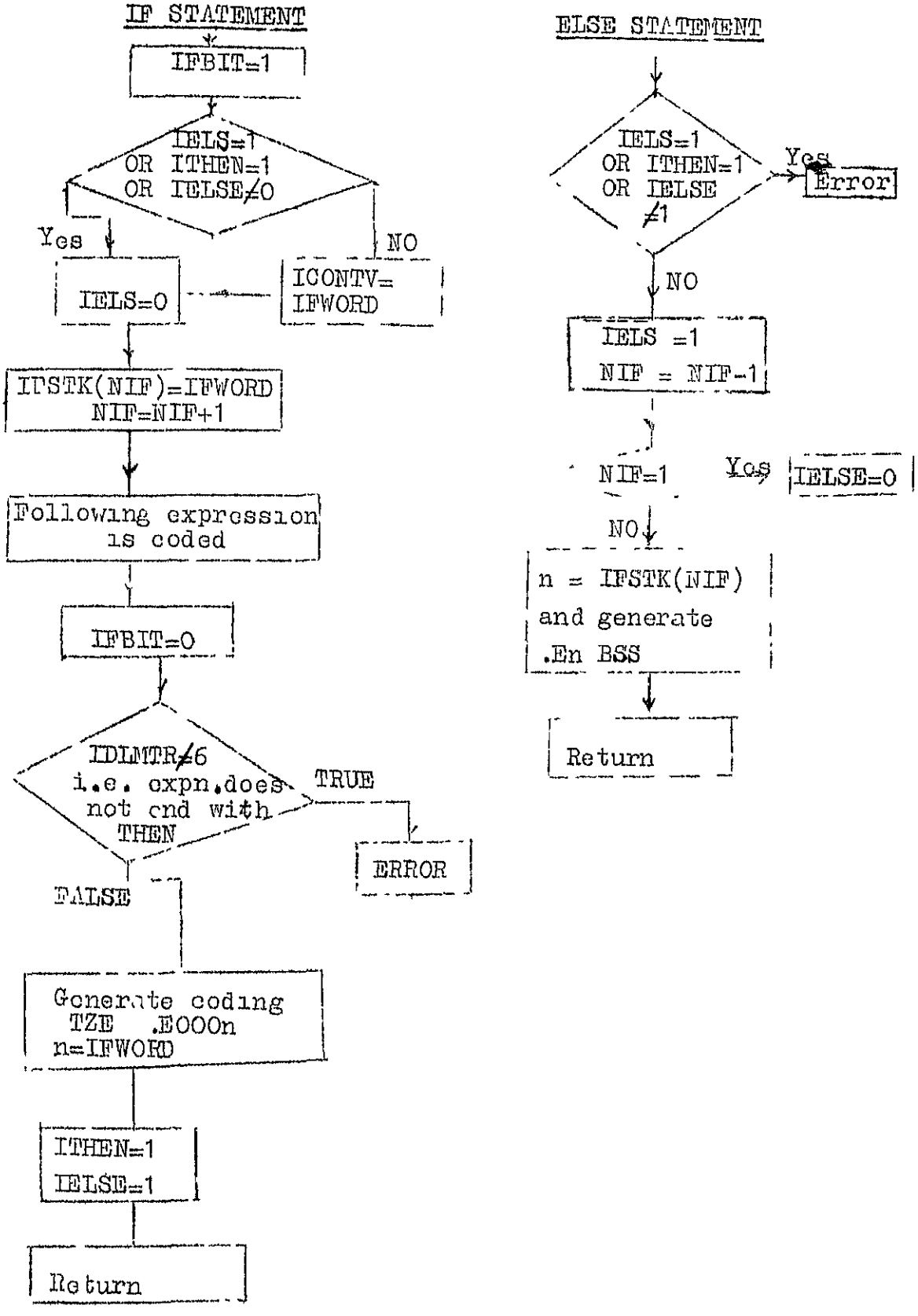
PROCDR: This routine is called, when any PROCEDURE statement occurs inthe source programme. It checks the syntax of the statement, puts the entry of procedure name in the symbol table and coding is done if the statement is found syntactically correct. In case of the parametric procedure and the function procedure, formal parameters are entered into the symbol table by calling the routine SYMTAB with LINK=0 and PPRNAM=1. Each time, external procedure is encountered, this routine erases all the symbol table entries except for the entry of the external procedure names and updates the pointers of the linkage of the keys having the same hash address accordingly.

MAIN: This is the main routine of the compiler. It finds the type of the statement and transfers control to the corresponding section of the programme either in "MAIN" routine itself or in other routines. Actually this routine consists of many subprogrammes. Besides this, programmes of intermediate processor and of Pass 2 are also submerged into this routine.

Here few routines (submerged in the "MAIN" routine) used in Pass I coding are described.

(1) BEGIN routine: This routine checks for the validity of the BEGIN statement. If it is found a valid one, the entry for it is created in the symbol table as well as in the block stack (BLOCK area). Block stack pointers are also advanced. If BEGIN is not a labelled statement an internal name is given to it and the entry for this name is created in the symbol table and in the block stack. Then Pass I coding (discussed in the next chapter) for this statement is done which works as a prologue of the block.

(2) GOTO routine: It checks the syntax of a GOTO statement. The label name of a GOTO statement is searched in the symbol table. If the label is found in the same block as of GOTO statement, it generates the transfer instruction for the GOTO statement after checking the validity of transfer. If the label is not found in the same block, it then searches the label in GOTO table. If this label is not found there also, it is put into the GOTO table. In both the cases a string is generated for the GOTO statement so that it may be coded in Pass II. In the next chapter GOTO and CALL statements are discussed in detail.


IF STATEMENT
IFBIT=1
IELS=1 OR ITHEN=1 OR IELSE≠0
Yes
NO
IELS=0
ICONTV= IFWORD
IFSTK(NIF)=IFWORD NIF=NIF+1
Following expression is coded
IFBIT=0
IDLMTR≠6 i.e. expn.does not end with THEN
TRUE
ERROR
FALSE
Generate coding TZE .E000n n=IFWORD
ITHEN=1 IELSE=1
Return
ELSE STATEMENT
IELS=1 OR ITHEN=1 OR IELSE ≠1
Yes
Error
NO
IELS =1 NIF = NIF-1
NIF=1
Yes
IELSE=0
NO
n = IFSTK(NIF) and generate .En BSS
Return

(3) GET/PUT routine: This is the I/O routine which checks the syntax of GET/PUT statements. Only EDIT directed I/O feature is allowed in IITPL. After syntactically found correct, coding for GET/PUT is also done in the routine. The coding of this statement is totally based upon the FORTRAN IV I/O routines.

(4) DO routine: This checks the syntax of DO statements and stores the corresponding information in the block stack (maximum of fifteen DO blocks can be active at a time). Two active DO blocks cannot have the same control variable. Full coding of the DO statement is done by this routine itself.

(5) IF/ELSE routine: It checks the syntax of IF & ELSE statements and keeps track of the nesting with the help of IFSTK table. By multiple nesting, maximum of twenty IFs' can be made active at a time. ELSE statement can only occur if a corresponding IF statement has already occured. All the active IFs' are terminated on encountering a non-ELSE statement.

(6) END routine: This routine checks the syntax of END statement. It physically ends the corresponding block or the group stack and moves back the pointers for the block/group stack. If any block ends, it generates coding for epilogue of this block. At the beginning of the coding a characteristic string is produced which helps in managing the block stack during Pass II. The coding for transfer - point to avoid those codings which are not part of the object programme, is also generated.

(7) RETURN routine: This routine, after checking the syntax of the RETURN statement, generates the coding for the termination of the procedure block in which it is contained. If any begin block(s) is present in the nested block stack, the coding for the termination of that (those) block(s) is also generated. In case of a function procedure, i.e. if the statement is of the form of "RETURN(expression),," the coding for the expression is generated in such a way that the value of the expression is put in AC (Accumulator) and the control is transferred to the calling point.

(8) OPEN/CLOSE routine: This routine checks the syntax of OPEN/CLOSE statements and generates coding which sets the indicator of the corresponding file ON/OFF respectively. When any file in I/O statement is referred, the indicator is tested. If this is on, it is supposed that the mentioned file is open otherwise error message is given during the execution.

Routines for handling the expression and assignment statemnt

ASSNRT: This is a syntax checking routine for the arithmetic expression. The routine is used for compilation of CALL, logical and arithmetic IF, DO, RETURN and arithmetic assignments. Basically this routine accepts as input the contents of COLECT area with its pointer denoted by "N" and prepares two stacks (in ISTAK) called the operand and operator stacks. While doing this it checks syntax and uses the SYMTAB routine to get the variable names or the function

procedure names in the symbol table. With the help of SYMTAB, ASSNRT differentiates the array name and the function name. ASSNRT stops preparing the stack whenever one of the followings is encountered:

(1) Semicolon (2) either of the four keywords THEN, TO, DO, WHILE (3) Zero level comma.

After successful preparation of the operator - operand stack, control is returned from ASSNRT and JMPTBL routine is called for the coding.

JMPTBL: This routine uses the stacks filled in ISTAK area by the ASSNRT for the coding. Operator jump table is shown which shows what action is to be taken for a particular operator-operator sequence. The main purpose of JMPTBL routine is to branch on the corresponding section of the programme to do the actions desired for a particular operator-operator sequence. The desired actions are also done in JMPTBL with the help of MODCHK routine and OPTMRT routine.

MODCHK: This routine checks the modes of two operands if possible during compilation. Otherwise it generates the coding for themode checking at the time of execution in case of statement function definition. It also sets the proper values to the pointers KOPRD, KPTR, IPTR1, IPTR2 (discussed later).

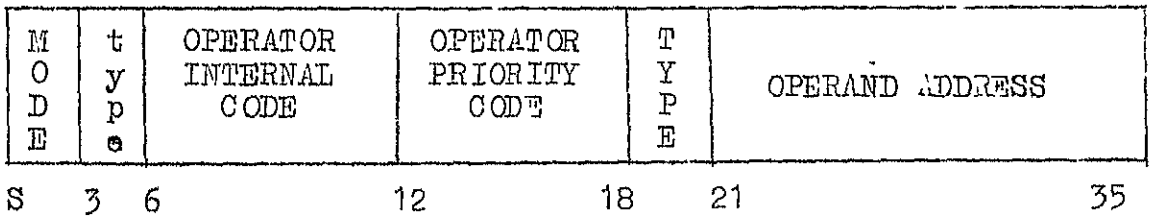
OPTMRT: This is the optimization routine, which optimizes the use of registers. This routine generates the coding for the forced operation.

Compilation Technique:

The method used in IITPL to compile the arithmetic expression or assignment statement involves two strings, one composed entirely of the operators in the same order as they appear in the original expression and the other composed entirely of the operands in the same order as they appear in the original expression. Null entry, which is denoted here by $\emptyset$, is included in the operator string so as to effect a correspondence between the two strings.

Representation of the Strings:

The two stacks of operand and operator have been compressed into a single string, i.e. to say that the ith entry in the operator string and the ith entry in the operand string occupy one IBM 7044 word in the format.



MODE = 0 for label variable
= 1 for integer
= 2 for bit-string
= 3 for character

type = 0 for simple variable
= 1 for 1-D array
= 2 for 2-D array
= 3 for 3-D array

TYPE = 0 if this operand is a local identifier
= 1 if it is a global identifier
= 2 if it is constant
= 3 if it is a function name.

Operator Priority Code	Operator internal code					
	ID=0	ID=1	ID=2	ID=3	ID=4	ID=5
1	≠					
2	=					
3	(					
4	⌈					
5	)					
6	⌉					
7	,					
8	+	-	Unary -			
9	*	/				
10	**					
11	.EQ.	.NE.	.GT.	.GE.	.LE.	.LT.
12	.NOT.					
13	.AND.					
14	.OR.					
15	.CAT.					

The operand address is the address of the symbol table, or a pointer to one of the stacks:

(i) IXTVAR (ii) ICONST (iii) IFUNC

The description of some of useful pointers are given below:

KOPRD: Address of that operand which will operate on the next operand.

IPT1: Pointer to the first operator in ISTAK.

IPT2: Pointer to the second operator in ISTAK.

KPTR: Current pointer to ISTAK which keeps the informa-

INITRG = 1 if initially AC is going to be busy.

= 2 if initially MQ is going to be busy.

IFINRG = 1 if finally AC is going to be busy.

= 2 if finally MQ is going to busy.

IPTR1 = pointer to ISTAK in which the operand corresponding to KOPRD is filled in ISTAK.

LREG = 0 if both registers are free.

= 1 if the last register used is AC.

= 2 if the last register used is MQ.

Example:

A = E = B*(C+D) - FUN(X)+SUB(E) ,.

The operator-operand stack for the above statement:

≠	ø
=	A
=	E
*	B
(	ø
+	C
)	ø
-	D
[	FUN
]	ø
+	X
[	SUB
]	ø
≠	I

IPT1/IPT2	≠	=	(	[	)	]	,	+-	*/	**	.RO.	.NOT.	.AND.	.OR.	.CAT.
≠	SPCL	SKIP	SKIP	SKIP	X	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP
=	EQUL	SKIP	SKIP	SKIP	X	X	X	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP
(	X	X	SKIP	SKIP	PERV	X	X	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP
[	X	X	FUNC	FUNC	X	FUNC	FUNC	FUNC	FUNC	FUNC	FUNC	FUNC	FUNC	FUNC	FUNC
)	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
]	ENDF	ENDF	X	X	ENDF	ENDF	ENDF	ENDF	ENDF	ENDF	ENDF	ENDF	ENDF	ENDF	ENDF
,	X	ERR	ARG	ARG	X	ARG	ARG	ARG	ARG	ARG	ARG	ARG	ARG	ARG	ARG
+-	ADSB	ERR	SKIP	SKIP	ADSB	ADSB	ADSB	ADSB	SKIP	SKIP	ADSB	SKIP	ADSB	ADSB	ERR
*/	MD	ERR	SKIP	SKIP	MD	MD	MD	MD	MD	SKIP	MD	SKIP	MD	MD	ERR
**	EXP	ERR	SKIP	SKIP	EXP	EXP	EXP	EXP	EXP	SKIP	EXP	SKIP	EXP	EXP	ERR
.RO.	REL	ERR	SKIP	SKIP	REL	REL	REL	SKIP	SKIP	SKIP	ERR	SKIP	REL	REL	SKIP
.NOT.	NOT	ERR	SKIP	SKIP	NOT	NOT	NOT	NOT	NOT	SKIP	NOT	ERR	NOT	NOT	NOT
.AND.	AND	ERR	SKIP	SKIP	AND	AND	AND	SKIP	SKIP	SKIP	SKIP	SKIP	AND	AND	SKIP
.OR.	OR	ERR	SKIP	SKIP	OR	OR	OR	SKIP	SKIP	SKIP	SKIP	SKIP	SKIP	OR	SKIP
.CAT.	CAT	ERR	SKIP	SKIP	CAT	CAT	CAT	ERR	ERR	ERR	CAT	SKIP	CAT	CAT	CAT

ADSB - is a common routine for addition and subtraction.

MD - is a common routine for multiply-divide.

.RO. - specifies any relational operator.

X - implies that the condition may never occur.

Processing of the Strings:

The technique used to process the strings is very simple and involves a two-way jump table (see the operator jump table) and two pointers IPT1 and IPT2 which point at the operator strings, with the help of the operator priority code and its value of ID (operator-internal code). Corresponding entry in the table is found and control is transferred to that entry.

Example: If IPT1 points at '≠' and IPT2 points '=' then the action indicated by 'SKIP' is performed. The definition of the table entries are given below:

(1) SKIP: IPT1 is set equal to IPT2 and IPT2 is advanced by one stage (ignore the null entries if any).

(2) RTWV: This occurs when IPT1 points at '(' and IPT2 points at ')'. The entries of IPT1 and IPT2 are erased, that is ISTAK (IPT1)=0 and ISTACK(IPT2)=0. IPT1 is retreated by one stage and IPT2 is advanced by one stage.

(3) ADD, SUBT, MPY, DVD, REL, AND, OR , NOT, CAT:

The operation specified by the pointer IPT1 in ISTAK is performed and the result is put in the entry for the 2nd operand in the operator string. The entry pointed by IPT1 is erased and IPT1 is retreated by one stage.

(4) FUNC: The initial linkage for the function procedure name (or the subscripted variable name) designated by the

1st operand is generated with the help of the two stacks (IICARD & ICSTK2). The first operand is erased and the forced operator [is replaced by ≠. Both IPT1 and IPT2 are advanced one stage.

(5) ARG: The coding for the subprogram argument (or subscript of the array) represented as the first operand, is generated using the two stacks mentioned above. The first operand and the forced operator are erased and both IPT1 and IPT2 are advanced one stage.

(6) ENDF: This terminates the generation of the function reference linkage (or subscript of the array name). The coding for the subprogramme argument (or subscript of the array name) represented as the second operand is generated and the complete coding for the function procedure (or array name) is moved from the stack IICARD to the object programme. The results is stored in temporary storage and the 2nd operand is replaced by that temporary storage. The forced operator and the first operand are erased. ITP1 retreated to the '≠' set up by (4) and it is erased. IPT1 is now retreated one stage and IPT2 is kept fixed.

(7) ERR: The operator indicated by IPT1 cannot be followed by the operator indicated by IPT2. Error message is given.

(8) EQUL : The equality is forced. IPT2 is kept fixed and IPT1 is retreated by one stage.

(9) SPCL: This finishes the compilation.

(10) EXP: Coding for exponentiation is generated. The forced operator is erased and IPT1 is retreated by one stage,

CODRTN:

This routine can be called from any where. It provides the coding to be written in correct MAP format. All the MAP instructions produced by the compiler, have, less than or equal to, three agruments and which always can be written in 30 columns of a card. That is why, all MAP instructions are written in only 5 words (IBM 7044). The format for invoking CODRTN is :

CALL CODRTN(IOPCOD, IARG1, IARG2, NUM1, NOARG),.

where IOPCOD - The OP-CODE of the instruction.

IARG1 - The first argument.

IARG2 - The second argument.

NUM1 - Number of characters in first argument.

NOARG - No. of arguments (1,2 or 3).

The variables linked by COMMON statement are

NUM3, IARG3, IOP(2), IAR(2), NDGT, ICALLS

where IARG3 - The third argument.

NUM3 - Number of characters in the third argument.

IOP(M) - Infix operator for linking some constant to Mth argument.

IAR(M) - Constant to be linked to the Mth argument by infix operator.

NDGT = 0 - Arguments 1 and 3 are variables

= 1 - Arguments are constants and the first argument should be preceded by prefix operator '='.

NDGT = 2 - Arguments are constant but there should not be any prefix operator '='.

ICALLS= 0 - Coding done in PASS I should be transferred to the buffer to write on tape 0.

= 1 - Coding should be done and kept in table.

= 2 - Statements should be taken from table and transferred to the buffer.

= 3 - PASS I endmark for one external procedure is given.

= 4 Coding done is in PASS II and should be transferred to buffer to write on tape 4.

If IOPCOD is not any valid MAP OP-CODE but is either of the five special characters ($\Delta, \backslash, \#, \vee, ';'$) then a string starting from COLECT(1) and of size 'NOARG' is transferred to the buffer of tape 0

CONVRT:

This routine converts the binary number into the alphabetic mode. The number sent as argument should not exceed the value 262143.

The format for invoking CONVRT routine is :

CALL CONVRT (N,M) ,.

where N= the number which is to be converted into the alphabetic mode.

M= 0 - The number sent will be left justified.

= 1 - The number sent will be right justified.

SETPTR:

It generates codings for EXTERN and ENTRY statements.

CHAPTER VI

PASS 1 CODING

Present chapter discusses about the actions done in Pass 1. It also describes about certain useful tables (BLOCK stack etc.) and pointers needed for compilation. Special attention is paid towards CALL and GOTO statements in Pass 1 besides the block structure.

PASS 1: Any IITPL source statement after being syntactically checked by the Syntax Analyser, is entered into this block for possible coding. Partial coding is also done for few statements, i.e. to say after a segment of the statement being found syntactically correct, corresponding coding is done. As for example let us consider the case of a DO statement. DO I =expn1 TO expn2 BY expn3.

Suppose expn1 is A+B. In such cases coding for expn1 is done and control variable I is set to its initial value equal to expn1, i.e. A+B in this case. The coding will be:

```
CLA  1,1
ADD  2,1
STO  3,1
```

where 1,2 and 3 show the corresponding relative position number of A, B and I in the present block. Then pointer is moved from "TO" to check the syntax of the remaining part of the statement. In such cases it is useful to do partial coding than to do whole coding after going through a full checking of syntax for the statement.

Similarly if a subscripted variable is declared, the corresponding coding is done.

Example: DECLARE A(5,10) FIXED ,.

the coding will be

```
.Vn     SXA     ..IDX2,2
        TSX     .SUBR2,2
        PZE     N
        PZE     5
        PZE     10
```

where 'n' is the internal number of subscripted variable and 'N' is the starting position of the subscripted variable. .SUBR2 is a runtime routine used to check the upper limit of subscripts.

BLOCK Structure:

A program segment which can contain local identifiers is called a block. Two kinds of blocks are provided for in IITPL (as said earlier).

(1) Procedure blocks (2) BEGIN blocks

The term "block" refers to either kind of block while its kind is indicated by the context.

Any procedure block except the Main Procedure block from where the execution starts, cannot be executed unless it is either properly called by another block or is referred as function. This calling or reference should, ofcourse, be a valid one, while BEGIN blocks are entered in the normal sequence of execution.

Consequently BEGIN block must be contained within another block which may be either BEGIN or Procedure block. The outermost block must be an external Procedure. The concept of "scope of identifiers" applies to BEGIN block also in exactly the same way as it applies to procedure blocks. However, BEGIN blocks cannot have parameters.

Before going into the detail of handling Block structure by compiler, few variable names used in the programme and their purposes are described below:

BLOCK: This is a single subscripted variable with upper limit of its subscript as 66. During compilation a block stack is formed and entered in BLOCK area. Nesting of blocks are allowed with upper limit as Seven (7). Each entry takes three words of IBM 7044 to keep the needed information. Entry for DO group has also been merged in this stack and a maximum depth of 15 (fifteen) for DO group is provided.

NB: It is a non-dimensioned variable and it points out the starting point of the current block or current group (whichever is last one) entered into the BLOCK area.

KBLK: This also is a simple variable and points out the starting point of the current block entered into the BLOCK area. Last block may either be BEGIN or Procedure block.

KPRO: This is a simple variable and points out the starting point of current procedure block entered into the BLOCK area.

IRPN: Internal relative position number: useful for giving the internal number (relative position number) to each identifiers (local or global) for the current block.

IBLNO: This represents the internal block number given by the compiler used during compilation as well as in execution (runtime routine). Each time a block is encountered, IBLNO is increased by one with its initial value as zero.

LSEQNO: This is the sequence number of BEGIN block or DO group. Each time DO group or BEGIN block is encountered LSEQNO is increased by one with its initial value zero.

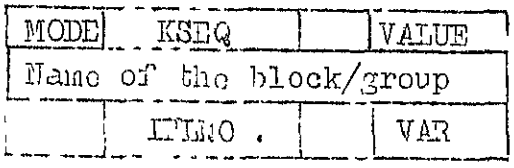
KSEQ: It is the current sequence number in case of BEGIN block or DO group. Else it is set to zero. It is used to check any illegal transfer in DO group.

FSL: Free storage location: used for preparing Symbol table. It is a single subscripted variable with upper limit of 4000.

DECBIT: It is set to 1 whenever new block is encountered and set to zero when any executable statement comes in the block.

BLOCK/Group Stack

As said earlier, each entry in Block stack takes three sconsecutive words of IBM 7044. BLOCK/group stack is formed in the following way:



MODE = 0 if the present block is procedure.

= 1 if it is a Begin block.

= 2 if it is a DO group.

KSEQ = 0 for procedure block

= current level sequence numb · in case of Begin block or DO group.

VALUE= pointer to the FSL (i.e. in Symbol table whether this block is entered).

= 0 for unlabelled DO statement.

If any unlabelled BEGIN statement comes, internal name is created and put into symbol table.

IBLNO= Internal block number (for this block).

= 0 for a DO group.

VAR: This is filled when any nested block starts, in such a case current block is entered into the BLOCK area and for previous block, VAR is set to IRPN. Then IRPN is set to zero to start with a new counting for present block.

Example:

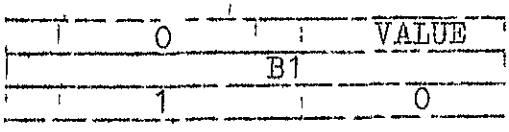
```
B1..PROCEDURE OPTIONS (MAIN) ,.
    DECLARE (A,B,C,D,E(6,4)) FIXED ,.
    A = B+C+D ,.
    ... ... ...
B2..BEGIN ,.
    DECLARE (E,F) FIXED ,.
    E = A+B+F ,.
B3..BEGIN ,.
    DECLARE (A,G) FIXED ,.
    A = A+B+C+D+E+F+G ,.
    END B3 ,.
D1..DO E = 1 TO 8 ,.
    F = E+F+B ,.
    END B1 ,.
```

When DECLARE E(6,4) FIXED ,. is encountered, IRPN will be 4 and the coding of the subscripted variable E(6,4) will be

```
.V0001 SXA     ..IDX2,2
       TSX     .SUBR2,2
       PZE     5
       PZE     6
       PZE     4
```

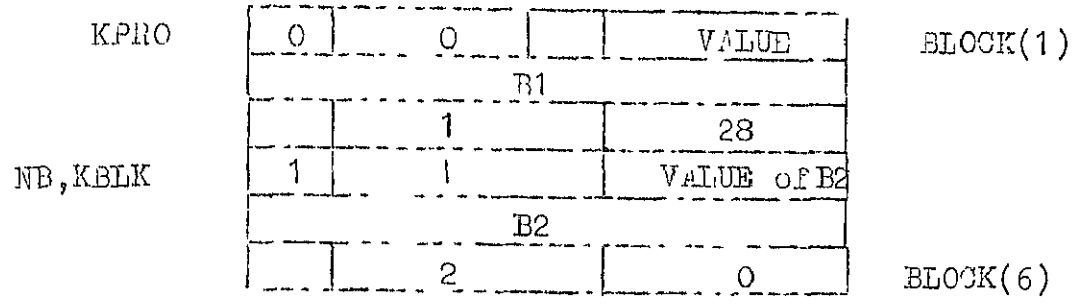
When B2..PROCEDURE ,. is encountered, IRPN = 28 and the BLOCK stack looks like

KPRO,KBLK,NB

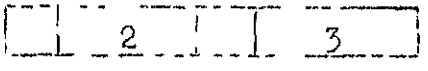


BLOCK(1)

Now value of IRPN will be put in $(BLOCK(3))_{21-35}$ and new block is entered like:



Similarly when B3..BEGIN ,. comes IRPN = 3 (two local identifiers (E and F) and one global identifier A). Consequently BLOCK(6) will have the contents as follows:

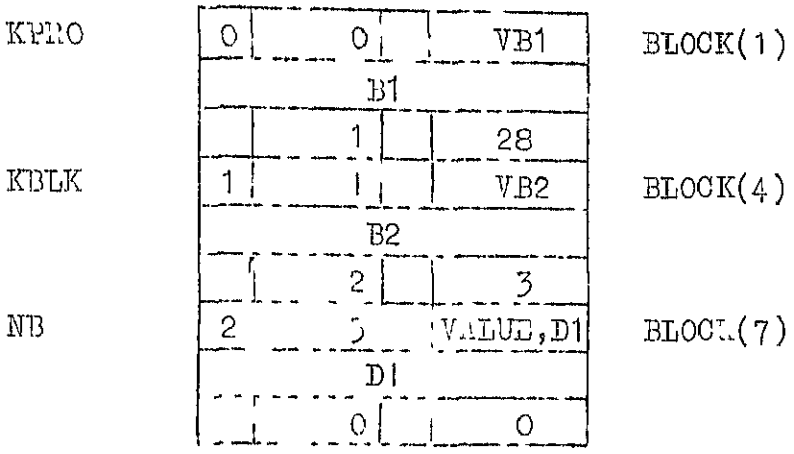


When END B3 ,. comes B = KBLK = 7, KPRO =1, IRPN =7 (two local A and G, three global needed from Block B1 (B,C,D) and two global from B2 Block (E and F)).

This IRPN will be put in $(FSL(VALUE+2))_{3-17}$ showing that the present block B3 needs only 7 locations to be used for identifiers during runtime. VALUE is the pointer to the symbol table where B3 is put.

Now entry for B3 is erased, pointers NB and KBLK are moved one stage back, i.e. KBLK = 4, NB = 4 and IRPN is set to the new value, i.e. $(BLOCK(KBLK))_{21-35}$ =3. So that if any other global identifier is needed, that will be given a relative position number 4.

When DO I = 1 to 8,. is encountered BLOCK entry looks like:



This is all about the block/group stack formed.

Possible Coding in Pass1 for blocks

Procedure block: If it is a Main Procedure and its name is, say, B1, the possible coding in Pass1 will be

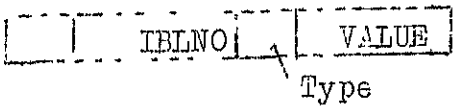
```
$IBMAP B1                    with DECBIT
       TRA    .In            set to 1
```

If any subscripted variable is declared in this procedure, the coding for the same will come after this.

As soon as any executable statement or any other block comes, following codings are done:

```
.In    BSS                   n denotes the corresponding block number.
XXXXXX
       TSL    .RTRTN
```

Where XXXXXX denotes a string of one word which looks like:



where TYPE = 0 for simple procedure block.

= 4 for parametric procedure or function procedure.

.RTRN - is a runtime routine used here for loading the current block into runtime stack.

External Procedure block i.e. with level 1: (but not a main procedure block:

P1.. PROCEDURE (A) ,.

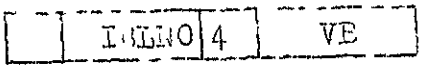
Codings:

```
          $IBMAP P1
                 ENTRY P1
          P1     TRA   *+
                 LXC   *-1,4
                 TRA   .In
```

(Coding for subscripted variable if any declared)

When any executable statement comes or another block comes, following codings are done.

```
.In   BSS
      XXXXXX  String of one word
```




```
      TSL     .RTRN
      TXI     *+n+1,,n      no.of argument
      PZE     MT1           mode and type of
                            1st arg.
```

```
        CLA     1,4
        PLT     1,4
        ANA     =Ø000000077777
        ANA     =Ø000060277777
        ORA     =Ø050000000000
        SLW     *+1
        CLA     **
        STO     arpn,1      arpn is the relative
                            position no. of 1st
                            formal parameter.
```

External Procedure (i.e. with Level 1) without any parameter:

```
        P2..PROCEDURE ,.
```

Coding:

```
        $IBMAP P2
               ENTRY P2
        P2     TRA   *+
               TRA   .In
        (Coding for Sub.Var.)
        .In    BSS
               XXXXXX
               TSX   .RTRTN
```

Internal Procedure Block (i.e. with level 1).

Example:

```
        P1..PROCEDURE OPTIONS (MAIN) ,.
        ...    ...    ...
        P3..PROCEDURE ,.
            Stmts
            END P2 ,.
            Stmts   .
```

Coding for P3..PROCEDURE ,. in Pass 1 will be

```
        TRA     .Pn
P3      TRA     **
        TRA     .In
        ...     ...     ...
.In     BSS
        AAAAAA  String      | IBL=0 | 0 |  VB |
        TSL     .RTRTN
        TSL     B.n
        ...     ...     ...
```

END P2,. is coded as:

```
B.n     TRA     **
(Coding for entry of global identifier)
        TRA     B.n
.Pn     BSS
```

Above example uses two label names:

(1) .Pn and (2) B.n

.Pn It is a label name which is put at the end of the coding of END statement for internal procedure block. This is done to maintain the normal sequence of execution of the programme.

In the above example TRA .Pn will transfer control to the next executable statement in P1 which is the normal sequence of execution since during execution control cannot enter into any procedure block without proper calling or proper reference.

B.n This is a label name put at the end of an internal block (procedure or begin) to make the entry for global identifiers needed in this block.

Coding for Begin Block:

B1..BEGIN ,, will be coded as

```
B1    TRA    .1n
...   ...    ...
.1n   TSL    B.n
```

Pass I: Coding of GOTO Statement

General format: GOTO L ,.

i) If the label 'L' is already present in the current block, then the code produced by the compiler is

```
      TRA    L
or    TRA    .1n      when there are
```

more than one labels with same name 'L' and n, internal number, is given to the label of this block.

ii) If the label 'L' has been declared as a label variable in the block, then the code is:

```
      TRA*   1,1
```

where '1' denotes internal position number of label variable in the variable stack.

iii) If label 'L' is not found in the block, statement is converted into a string as PASS I output which in turn is used by PASS II for proper coding.

Example:

GOTO L ,. The corresponding string for this statement will be: 00000++ 000001 L⌀⌀⌀⌀⌀ -01000

First word is the string characteristic word.

If the label is not found even in the current procedure (not block), it is filled in GOTO table. The management of GOTO table is done with the help of IGRLK table and its pointer NPB (see appendix).

CALL Statement

Due to possibility of occurence of inter-subprogram transfer GOTO statements, the coding of CALL statement is never possible in PASS I. The required return points (needed by inter-subprogramme GOTO statement) will be supplied with the coding expansion of CALL statement in PASS II.

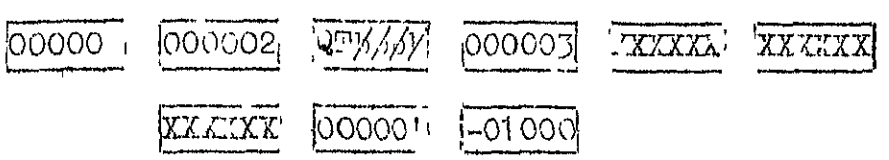
Expressions to be transferred as arguments of the CALL statement are coded in PASS I itself. The value of the expression are kept in runtime temporary location.T.+n and its mode in ..T+n respectively. Where 'n' is the next available temporary location.

The CALL statement is converted into a string as PASS I output, which in turn is used by PASS II for the proper coding.

Example:

i) CALL P,. - 00000+ 000001 P⌀⌀⌀⌀⌀ 000000 -01000

11) CALL QT(M,S+T*U, 11) ,.



(a) First word is the string characteristic word which is used for recognizing the string in PASS II.

(b) Second and third words keep the characteristic and name of the 'Called Procedure'.

(c) Fourth word keeps the number of arguments of the CALL statement.

(d) The word shown by XXXXX in the string is the CALL Argument characteristic Word. The details of the informations stored in the 'Call Argument Characteristic Word' are discussed in the Appendix II.

CONCLUSION

Like FORTRAN compiler, IITPL compiler is also an out-core compiler, except the Runtime routines, which are there, in the core, at the time of execution. The Runtime routines approximately take 900 memory locations. Besides this, the compiler takes the help of some FORTRAN routines (namely I/O routines, EXP1 ...) which also will occupy some memory space but the memory, occupied by these routines are programme dependent. Only those routines will be in the core which are required by the programme, otherwise they will remain outside.

The compiler output is an IAP (assembly language of IBM 7044), hence the assembler and its IOCS occupy round about 6000 memory locations. So, out of total 32768 memory locations, approximately 25000 memory locations are free for the object programme.

Since the compiler is an out-core type, its symbol table is so designed that it causes the compiler to take the whole of memory. If some addition is to be done in the compiler, the size of the symbol table (which is at present 4000) can always be reduced with a slight change in the system.

All the three sections, Lexical, Pass I & Pass II, and Runtime routines, have been tested separately and successfully. Till the end, the programme, handling CALL & GOTO statements, in Pass II, could not be fully tested, hence all the three sections of the compiler were not combined.

APPENDIX I

Here an example is taken to illustrate the PL/S I output of the source statement (IITPL).

```
*PL/1
P1..PROCEDURE OPTIONS (MAIN) ,.
    DECLARE (A,B,C,D,E,I) FIXED ,.
    DECLARE F(5,6,7) FIXED ,.
 M..A=A+B ,.
 L..IF(A.EQ.B) THEN IF(C.GT.2) THEN IF(A.NE.C)THEN A=0 ,.
    ELSE ,.
    ELSE IF(A.LT.8) THEN C=5 ,.
    ELSE A=C ,.
    CALL P2(A) ,.
    DO I=1, D WHILE A=3, 4 TO 7 BY 3 ,.
    A=A*(B+C/D**5)-8 ,.
    END/* THIS SHOWS THE END OF DO GROUP */ ,.
B1..BEGIN ,.
    DECLARE (A,M,N,P) FIXED ,.
    A=A+B ,.
    END B1 ,.
    GET EDIT (C,D,E) (F(6), X(3)) ,.
P2..PROCEDURE (G)/* SECOND BLOCK STARTS */ ,.
    DECLARE G FIXED ,.
    GOTO M,.
    RETURN ,.
    END P2 ,.
```

```
G.. PUT A((3 F(6),X( )),.
    PUT EDIT(('F(',C) DO A=D TO E
    BY 1) DO F=3 TO E) DO G=1 TO 6
    BY 3) (F(G)) ..
    END P1 ,.
    'DATA
```

Codings:

```
P1..PROCEDURE OPTIONS (MAIN) ,.
    DECLARE (A,B,C,D,E,I) FIXED ,.
    DECLARE F(5,6,7) FIXED ,.
```

```
$IBMAP   P1
P1       IRA    .IC00'
.V00C1   SXA    ..IDX2,2
         TXI    .SUBR3,4
         PZE    -
         PZE    5
         PZE    6
         PZE    7
.L0001   BSS
00000    XXXXXX   A string which looks like
                  | TRUNC | 0 | VALUE |
                  5        17  21      35
         TSL    .RTRTN
```

```
M..A=A+B,.
M        CLA    1,1
         ADD    2,1
         STO    1,1
```

```
L..IF(A.EQ.B) THEN IF(C.GT.2) THEN IF(A.NE.C) THEN A=0,.

                L       CLA     1,1
                        SUB     2,1
                        TSX     RO.EQ.,4
                        TZE     .E0001
                        CLA     3,1
                        SUB     =2
                        TSX     RO.GT.,4
                        TZE     .E0002
                        CLA     1,1
                        SUB     3,1
                        TSX     RO.NE.,4
                        TZE     .E0003
                        CLA     =0
                        STO     1,1
                        TRA     .F0001
ELSE,.
ELSE IF(A.LT.8) THEN C=5,.
ELSE A=C ,.
ELSE ,.
                .E0003  TRA     .F0001
                .E0002  BSS
                        CLA     1,1
                        SUB     =8
                        TSX     RO.LT.,4
```

```
                    TZE     .E0004
                    CLA     =5
                    STO     3,1
                    TRA     .F0001
             .E0004 BSS
                    CLA     3,1
                    STO     1,1
                    TRA     .F0001
             .E0001 TRA     .F0001
             .F0001 BSS
```

```
CALL P2(A) ,.
```

000007	000002	P2ØVØØ
000001	XXXXXX	-01000

Where 'XXXXXX' is argument characteristic word.

```
DO I = 1,D WHILE A=3,4 TO 7 BY B ,.
A=A*(B+C/D**5)-8 ,.
END /* THIS SHOWS THE END OF DO GROUP */ ,.
                    CLA     =1
                    STO     6,1
                    TSL     .D0001
                    CLA     4,1
                    STO     6,1
                    CLA     1,1
                    SUB     =3
                    TSX     R0.EQ.,4
                    TZE     *+2
```

```
          TSL     .D0001
          CLA     =4
          STO     6,1
          CLA     =7
          STO     D.0001+3
          CLA     2,1
          STO     D.0001+4
.W0001    TSL     .D0001
D.0001    TSX     .DORTN,4
          CLA     6,1
          STO     6,1
          PZE     0
          PZE     0
          TZE     E.0001
          TRA     .W0001
.D0001    TRA     *+1
```

* Coding of next statement starts here i.e. Assignment statement in this case.

```
          LDQ     4,1
          MPY     4,1
          STQ     .T.+0
          MPY     .T.+0
          MPY     4,1
          STQ     .T.+0
          LDQ     3,1
```

```
          PXD     ,0
          DVP     .T.+0
          LLS     35
          ADD     2,1
          LRS     35
          MPY     1,1
          LLS     35
          SUB     =8
          STO     1,1
      * Coding of END statement
          TRA     .D0001
       E.0001 BSS
B1..BEGIN ,.
   DECLARE (A,M,N,P) FIXED ,.
   A=A+B ,.
   END B1 ,.

       00000    XXXXX      A string which looks like

                           |  |  IDLNO  |O|VALID  |
                              3       17 21      35

       B1      TRA     .L0002
       .L0002 BSS
               TSL     B.0002
               CLA     1,1
               ADD*    5,1
               STO     1,1
       00000;
```

```
                    CLA     =2
                    TSL     .RTRTN
                    TRA     .B0002
           B.0002   TRA     **
                    CLA     =Ø000002000005
                    TSL     .RTRTN
                    CLA     =3
                    LDQ     =1
                    TSL     .RTRTN
                    ADD     =2
                    STO     5,1
                    TRA     B.0002
           .B0002   BSS
GET EDIT(C,D,E) (P(6),X(5)) ,.
                    TSL     STFDX
                    TSX     TSHIO.,4
                    TWO     FILO5.
                    PZE     nS
                    TSL     RESET.
                    TSL     HNLIO.
                    STO     3,1
                    TSL     HNLIO
                    STO     4,1
                    TSL     HNLIO.
                    STO     5,1
                    TSX     RTNIO.,4
                    TRA     mE
           nS       TSX     IOHIC.,4
                    PZE     6
```

```
                  PZE     3
                  TRA     IOHEF.
          mE      BSS                  where 'nE' is internal
                                       name given by the
                                       compiler, and 'm'=n+1
P2..PROCEDURE(C) ,.
    DEFINING C FILED ,.
                  TRA     .T0003
          P2      TS      *,
                  CAL     *-1,4
                  TRA     .T0003
       .T0003     BSS
       000004     XXXXXX   string similar as mentioned
                           earlier.
                  TSL     .TATN
                  TXI     *+2,,1
                  PZE     1
                  STA     1,4
                  PAT     1,4
                  ANA     =Ø000000077777
                  ANA     =Ø000060277777
                  ORA     =Ø050000000000
                  SLW     *+1
                  CL..    **
                  STO     1,1
                  TSL     B.0003
GOTO M ,.
       000004  000001  MØØØØØ
```

```
RETURN ,,
                    TRA     A.0003
END P2 ,,
          00000;
          A.0003 CL..     1,4
                 PLT      1,4
                 ANA      =Ø0000000077777
                 ANA      =Ø0000077777777
                 ORA      =Ø050000000000
                 SLW      *+2
                 CLA      1,1
                 SBO      **
                 CLA      =2
                 TSL      .RTRTN
                 TRA      P2
          B.0003 TRA      **
                 TRA      B.0003
          .P0003 BSS
G..FORMAT(3 F(6),X(1)) ,.
                 TRA      nE
          G      AXT      3,2
                 TSX      IOHIC.,4
                 PZE      6
                 TSX      IOHXC.,4
                 PZE      1
                 TRA      IOHEF.
          nE     BSS
```

```
PUT EDIT(((F(A,B,C) DO A=D TO E BY I) DO B=3 TO E) DO C=I TO 6
                                   BY 3) (R(G)) ,.

                TSL     STNDX.
                TSX     STHIO.,4
                TWO     FILO6.
                PZD     2S
                TSL     RESET.
                LDQ     6,1
                STQ     3,1
.S0002          BSS
                LDQ     =3
                STQ     2,1
                CLA     5,1
                PAX     0,4
                SXD     .S0003,4
.S0004          BSS
                LDQ     4,1
                STQ     1,1
                CLA     5,1
                PAX     0,4
                SXD     .S0005,4
                CLA     6,1
                PAX     ,4
                SXD     .T0003+2,4
                LDQ     3,1
                VLM     =983040,,15
```

```
        ALS     15
        LDQ     2,1
        VMA     =153640,,15
        ADD     1,1
        SUB     ..IDX1
        PAC     ,2
        CLA     6,1
        PAX     ,4
        SXD     .N0001,4
.S0006  BSS
        TSX     .V0001,4
        CLE     1,1
        ONE     2,1
        ONE     3,1
        CLA     -29,2
        TSL     UNLIO.
.N0001  TXI     *+1,2,
.T0003  CLA     1,1
        PAX     0,4
        TXI     *+1,4,*-
        PXA     0,4
        STO     1,1
.S0005  TXL     .S0006,4,**
.T0002  CLA     2,1
        PAX     0,4
        TXI     *+1,4,1
        PXA     0,4
```

```
                    STO     2,1
            .S0003  TXL     .S0004,4,**
            .T0001  CLA     3,1
                    PAX     0,4
                    TXI     *+1,4,3
                    PXA     0,4
                    TXL     .S0002,4,6
                    TSX     RTNIO.,4
                    LXA     ..IDX2,2
            2S      EQU     6
END P1 ,.           OOOOO;
                    TRA     S.JXIT
                    ENTRY   ..VAR
                    EXTERN  ..IDX2
                    EXTERN  .RORTN
                    EXTERN  .RTRTN
                    EXTERN  .SSUBR3
                    EXTERN  .DORTN
            ..VAR   BSS     223
            .T.     BSS     1
                    END     P1
            $ENTRY
```

Note:

.SUBRm - This is the runtime routine for 'm' subscripted variable. It checks whether the subscripts exceeds its upper limit or not. If any one of the 'm' subscripts exceeds its upper limit, it gives error in execution time.

.RORTN - This is the relational operator runtime routine which sends zero in AC if the condition is false otherwise it puts 777777777777 in AC.

.RTRTN - This routine does the following:

i) if C(AC) is greater than 3, then it assumes that contents of Accumulator is in the following way:

Block No.	Type	No.of variables needed

If Type = 1 - then it loads the current block in the active block stack (ACTBLS) and moves the pointer (Index1) correspondingly.

= 4 - It assumes that current block is a parametric procedure block or functional procedure block. It checks the modes of the arguments and loads the current block.

ii) If the C(AC)=3 - it assumes that block no. has been sent in MQ. It then takes out the starting pointer in the variable stack (..VAR) for this block and puts it in AC and returns to the calling point.

iii) If the C(AC) = 2 - it erases the current block entry from ACTBLS (Active Block Stack).

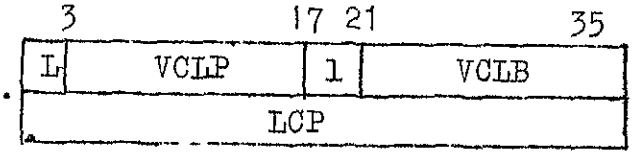
.DORTN - This routine checks whether the control variable has reached its limit value or not. If not it sends'1' in AC otherwise puts zero in AC and returns.

..IDX2 - This saves the content of index2.

APPENDIX II

ICALL Table:

In PASS I, when any procedure is called from any block; level and the symbol table address of the "Called Procedure" and that of "Calling Block" is kept in a pair of words of ICALL table as shown below:



where

L - Level of "Called procedure".

l - Level of "Calling block".

VCLP - Symbol table address of "Called procedure".

VCLB - Symbol table address of "Calling block".

LCP - Label name of "Called procedure".

It is not always possible to find the "Called procedure" in symbol table, in that case $C(\text{first word})_{S-17}$ is filled with zero. When a PROCEDURE statement is encounter d in PASS I, it is checked whether the name of the new procedure appears any where in the ICALL table undefined. If yes then L and VCLP is filled at that time.

Before filling information in the ICALL table, following things are kept in mind:

i) Repetition is avoided.

ii) (a) If "called procedure" is already in symbol table

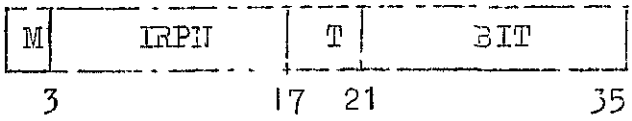
the following condition is to be satisfied.

L-1 = 0 or 1

(b) If "called procedure" has not come atall before the appearance of CALL statement, the validity of above condition is checked when PROCEDURE statement with the label of "Called procedure" appears.

One hundred locations are reserved for ICALL table and the variable "NOCALL" is used as pointer of ICALL table.

CALL Argument Characteristic Word



where IRPN = Internal position number of the variable or temporary (when the argument is expression).

BIT = Number of bits required in the case of bit string and character string, otherwise 0.

T = 0 simple variable
= 1 Single subscripted
= 2 Double subscripted
= 3 Triple subscripted
= 4 Temporary (in case of expression)
= 5 Constant
= 6 Functional argument

When T=5 'IRPN' points the number of words following the characteristic word which store the constant.

M = 0 Label variable

= 1 Simple and subscripted variable

= 2 Bit string variable

= 3 Character string variable

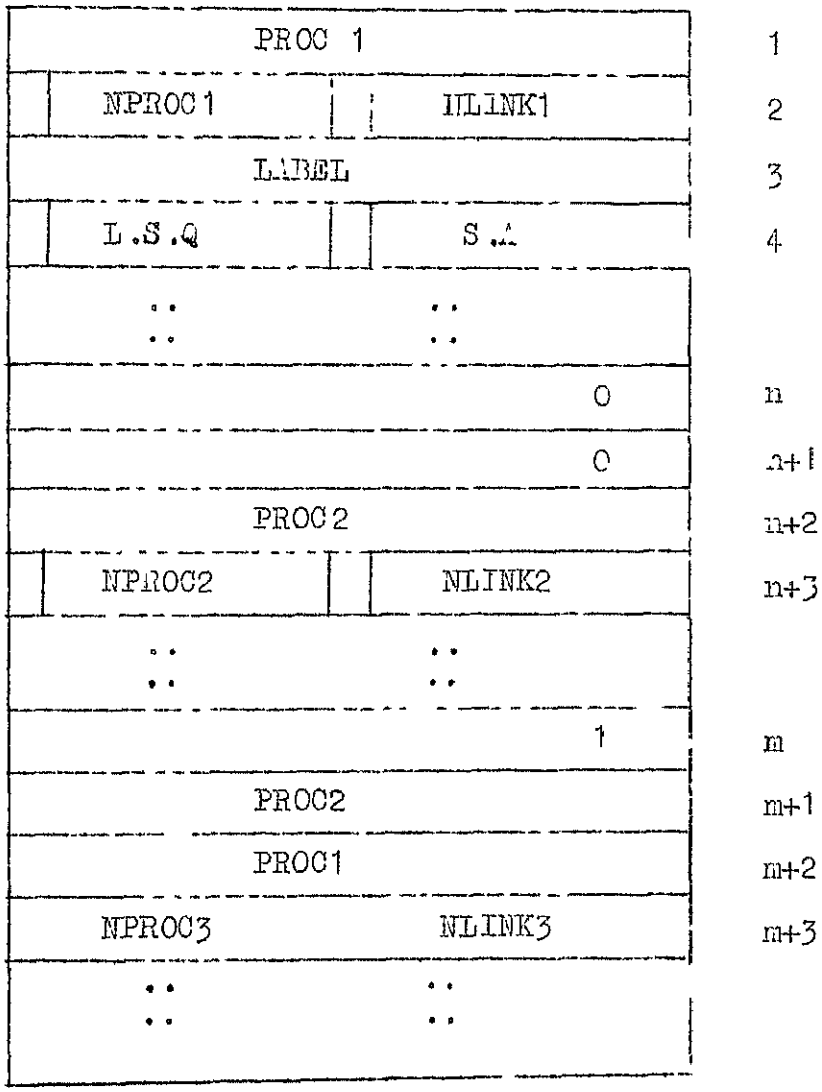
= 5 External fixed variable

= 6 External bit variable

= 7 External character variable

GOTO Table

GOTO table is is the table of unfound labels of procedures. All unfound labels of any particular procedure are kept as an element of the string, whose HEADER keeps the name of the procedure and information about the next link of the string.



where NPROC - Table address of the first element of the string, which starts just after this string section is over.

NLINK - Table address of the HEADER of the next link of the very PROCEDURE string.

SA - Symbol table address of the label.

L.S.Q - Level Sequence-number

Before storing the unfound label and its characteristics, it is always checked whether it is already an element of the string of unfound labels of the procedure.

If END statement corresponding to any procedure is encountered and if this procedure is having a string of unfound labels, then it is permanently closed with storing 1 and procedure's label in the last two consecutive locations of the string of the GOTO table and running procedure is rubbed off from IGBLK table.

If any procedure starts before permenent end of the last running procedure, then the GOTO table string of the previous procedure is temporary closed with 0 and 0 in next two free locations and then HEADER for new string is kept thereafter.

As shown in example above:

NPROC1 = Table address of PROC2 i.e. n+2.

NLINK1 = address of next link of PROC1 i.e. n+2

NPROC2 = Table address of next procedure(i.e. of PROC1) i.e. n+2.

NLINK2 = 0

IGBLK Table:

This is the procedure block table used for handling the GOTO table. Each time PROCEDURE statement is encountered, it is entered in the IGBLK table. Each entry in IGBLK requres two words of IBM 7044 which consists the following:

Procedure name
SPTGO

where 'SPTGO' is the starting pointer in the GOTO table wherefrom any external label needed for this procedure block is put. The first entry in the GOTO table for any procedure block is the procedure name itself and the string of unfound labels, needed by that particular procedure, are put thereafter. If there is no global label needed by this procedure block, SPTGO is set to zero.

'NPB' is the pointer for handling IGBLK table. When an END statement is encountered the 'NPB' is moved up.

APPENDIX III

LEXICAL ERROR LIST

ERROR NO-1 Presence of illegal character in the statement.

NO-2 Either integer with more than 11 digits or binary string of more than 36 elements or character string of more than 36 characters, found in the statement.

NO-3 Identifier, with first character as digit, encountered.

NO-4 Identifier of more than six characters or wrong logical word, or keywords used as variable.

NO-5 Arithmetic expression with unbalanced parentheses.

NO-6 The first element of the statement is an operator.

NO-7 More than one continuation cards, encountered.

NO-8 Non-binary element found in the binary string.

BIBLIOGRAPHY

1. IBM System/360 - PL/1 Reference Manual Form No.C28-8201-1 IBM System Reference Library.

2. Programming Language/OPT by Frank Bates and Mary L. Douglas, Prentice Hall Inc., 1967.

3. Compilation Technique, D.R.A. Hopgood.

4. Scatter Storage Techniques, CACM, January 1968, by Rober Morris.

5. WATFOR Compiler CACM, January 1967, by W.Shantz, R.A.German, J.G. Mitchell, R.S.K.Shirtley, C.R.Zanke.

E-1971-M-SIN-LEX